Lossless Compression of Double-Precision Floating-Point Data for Numerical Simulations: Highly Parallelizable Algorithms for GPU Computing

Mamoru OHARA^{†a)} and Takashi YAMAGUCHI[†], *Members*

SUMMARY In numerical simulations using massively parallel computers like GPGPU (General-Purpose computing on Graphics Processing Units), we often need to transfer computational results from external devices such as GPUs to the main memory or secondary storage of the host machine. Since size of the computation results is sometimes unacceptably large to hold them, it is desired that the data is compressed and stored. In addition, considering overheads for transferring data between the devices and host memories, it is preferable that the data is compressed in a part of parallel computation performed on the devices. Traditional compression methods for floating-point numbers do not always show good parallelism. In this paper, we propose a new compression method for massively-parallel simulations running on GPUs, in which we combine a few successive floating-point numbers and interleave them to improve compression efficiency. We also present numerical examples of compression ratio and throughput obtained from experimental implementations of the proposed method running on CPUs and GPUs.

key words: GPGPU, massively-parallel numerical simulation, data compression, double-precision floating-point data

1. Introduction

Massively parallel simulation has become a commonly used scientific methodology with the popularization of COTS chip multiprocessors and coprocessors. Today, we can achieve billions of FLOPS by interconnecting PCs containing multicore CPUs with high-speed LANs. Moreover, even a PC employing a dedicated graphic processing unit (GPU) can gain over 500G FLOPS in some parallel applications by utilizing the dedicated unit for general purpose processing, that is called GPGPU (General-Purpose processing with GPU).

In numerical simulations using massively parallel computers like GPGPU or dedicated hardware accelerators, which are sometimes implemented in FPGAs (Field Programmable Gate Array), we often need to transfer intermediate computational results from external devices such as GPUs to the main memory or secondary storage of the host machine in order to display them or save them for checkpointing [1], [2]. Since the scale of simulations expands along with the growth of computing power, the size of the computation results is sometimes unacceptably large to hold them. Therefore, it is desired that the data is compressed and

stored; this is especially true for checkpoint data, which is used only when failures occur. Considering overheads for transferring data between the devices and host memories, it is preferable that the data is shrunk in the devices. For example, we can implement compressor circuits for simulation data in the same FPGA in which we also implement accelerator circuits. Some techniques for implementing the compressor in FPGAs have been proposed [3], [4]. On the other hand, numerous lossy compression algorithms for graphics data running on GPUs have been reported, however, there are only a few ones for simulation data. We in this paper propose some GPGPU algorithms for compressing simulation results in GPUs.

For numerical simulations, compression must be lossless. Moreover, most data of numerical simulations are double-precision floating-point (FP) numbers. Thus resulting data from the simulations usually consists of many double-precision FP numbers. In literature, a few but not so many lossless compression techniques for FP data have been reported. Some of the compression techniques are designed for specific use: Isenburg et. al. presented an arithmetic-coding-based method for compressing 3D geometric data [5]–[7]. Harada et. al. developed compressors for MPEG-4 audio data [8]. They are not always suitable for compressing resulting data of numerical simulations.

FPC is one of general-purpose lossless compression techniques for double-precision FP numbers [9] (hereafter we simply refer to a double-precision FP number as double). FPC can well compress FP data of wide applications, such as message passing, satellite observation, and numerical simulations. FPC deploys a 1-pass compression algorithm for performance. It predicts a value in a data stream based on a hysteretic technique and gets the difference between the true and the predicted values. The difference is likely to contain many zero-valued bits when the prediction is well performed. FPC compresses the leading zero-valued bytes in the difference by encoding the number of the leading zero bytes (NLZ) into a few bits. It is expected that encoding NLZ is more preferable for 1-pass compression than entropy coding or dictionary-based methods.

Conventional 1-pass prediction algorithms such as that used in FPC have some disadvantages for using in massively parallel numerical simulations. Most of all, these 1-pass algorithms are designed for serial processing, thus they are not easy to implement them on massively-parallel comput-

Manuscript received January 10, 2012.

Manuscript revised June 7, 2012.

[†]The authors are with the Tokyo Metropolitan Industrial Technology Research Institute, Tokyo, 135–0064 Japan

a) E-mail: ohara.mamoru@iri-tokyo.jp

DOI: 10.1587/transinf.E95.D.2778

ers. Although Burtscher et. al. developed a parallel version of FPC, pFPC [10], it is suitable for computers having some multi-core processors rather than massively-parallel systems. Therefore, recently O'Neil and Burtscher proposed another parallel FPC named GFC whose prediction algorithm is thoroughly redesigned for GPGPU [11]. While GFC exceeds pFPC in compression throughput on GPUs by about five times, its mean compression ratio is much worse than that of pFPC.

In addition, precise prediction of FP values is essentially difficult. This is especially true for 64-bit double data. Usually, we can obtain only small number of leading zero bytes in FP data, and thus we cannot take full advantages of NLZ-based methods.

In this paper, we propose a highly-parallelizable lossless compression technique for double data resulting from numerical simulations. It is expected that a resulting data stream from a numerical simulation has few discontinuous points and varies smoothly, thereby we suppose we can predict the values by applying interpolating polynomials. Similar to some previous work in arithmetic predictors [3], [4], [12], we construct a simple and highly-parallelizable predictor based on the Newton polynomial. We can improve parallelism of a compression technique by using simple arithmetic predictors. This approach is similar to GFC, however, by combining two different polynomials in the same manner as FPC, accuracy of our predictor is comparable to FPC and much higher than that of GFC.

We also propose a recombination technique in which we combine and interleave bytes of several FP numbers in order to improve compression ratio. There are two portions in a FP number in terms of predictability: while a sign bit, an exponent, and a couple of higher bits of mantissa are relatively predictable, the lower bits of mantissa are very hard to predict. We combine four doubles and rearrange the byte order of the doubles so that the relatively-predictable portions are gathered in front in order to make long NLZ. It is expected that by the rearrangement, we can efficiently compress the predictable portions with almost no negative effects on compression of unpredictable portions, which are not compressible so much anyway.

We implement both CPU and GPU versions of the proposed technique and measure their compression ratio and compression throughput for several data sets resulting from numerical simulations. We found that the proposed technique shows comparable compression ratio to FPC and pFPC and can improve its compression throughput by using a GPU.

The remainder of this paper is composed as follows: 2 introduces some conventional compression techniques for FP data including FPC, pFPC, and GFC and discusses the predictability and parallelism of these techniques. 3 and 4 propose the prediction algorithm based on the Newton polynomial and the recombination algorithm, respectively. Some numerical examples of compression ratio and throughput of our algorithms, FPC, pFPC, GFC, and some traditional compression techniques are presented in 5. Fi-

nally, 6 concludes this paper.

2. Preliminaries

In this section, we discuss some compression methods for FP numbers. In various types of processors and softwares including CPUs and GPUs, the FP number formats standardized in IEEE 754 [13] are used. We suppose that most applications of numerical simulation also adopt IEEE 754 format. Therefore, in this paper we will present a technique for compressing floating-point data based on IEEE 754.

A FP number based on IEEE 754 has a sign-bit followed by l_e -bit-length exponent and l_m -bit-length mantissa. IEEE 754 defines a number of formats which have different widths in bits of these fields: that are, binary16 (half precision: $l_e = 5, l_m = 10$), binary32 (single precision: $l_e = 8, l_m = 23$), binary64 (double precision: $l_e = 11, l_m = 52$), binary128 (quadruple precision: $l_e = 15, l_m = 112$), and some decimal formats. In scientific applications, binary64 format, so-called double-precision FP number, is widely used, thus, we in this paper mainly focus binary64 format. While the order of the 8 bytes of a double held in memory differs according to processor architectures, in this paper, we always locate the sign bit at the highest position, and the exponent and fraction parts follow as shown in Fig. 1. The exponent field is biased by 1023 for omitting a sign bit for exponent; mantissa is also normalized so that its integer part is always 1 and only the fraction part is held in the fraction field.

For compressing resulting data of scientific applications, we need lossless compression techniques. In literature, a few but not so many lossless compression techniques for FP data have been reported. Isenburg et. al. presented an arithmetic-coding-based method for compressing 3D geometric data [5]–[7]. In their method, we first predict coordinates of a point in a 3D mesh model, each of which is expressed in FP numbers, from coordinates of the adjacent points by some techniques, typically by parallelogram predictors. Then, we compress the difference between the true value and the predicted value with range coders [14], [15]. Although this technique may be applicable to numerical simulations if it is used in combination with appropriate prediction techniques, the authors did not refer particularly to such techniques.

Harada et. al. also proposed a compression technique specialized for audio data stream of MPEG-4 Audio Lossless Coding (ALS) [8]. Since digital audio data is inevitably quantized in analog-digital conversion, floating-point numbers are not usually used in audio coding for personal use.

Fig. 1 The format of IEEE 754 binary64 (double-precision floating-point numbers).

In professional audio applications, floating-point numbers are used mainly for avoiding overflow and underflow in intermediate processes and thus floating-point audio data is often derived from quantized integer numbers. Based on this, Harada et. al. separate audio FP data into multiples of an integer and error fraction, and then compress the integer and fraction components by different methods.

FPC is one of prediction-based lossless compression techniques for general-purpose FP applications [9]. FPC employs a hybrid predictor of FCM (finite context method) [16] and DFCM (differential FCM) [17]. While FCM is originally designed for text compression, it is revamped into a bit-stream-based predictor in order to fit for 1-pass compression of FP data in FPC. That is, for sequential inputs of FP data, the predicted value is calculated by the psuedo codes below:

```
double v;
uint64 u, pred, hash, fcm[M];
u = bit_pattern_of(v);
pred = fcm[hash];
fcm[hash] = u;
hash = ((hash<<6) XOR (u>>48)) MOD M.
```

where uint64 is the data type of 64-bit unsigned integer and variables u and v have the same bit pattern. Moreover, fcm is a table whose size M is a given parameter. On the other hand, DFCM records not the value itself but differentiates two successive values, i.e.,

```
double v;
uint64 u, last, pred, dhash, dfcm[M];
u = bit_pattern_of(v);
pred = dfcm[dhash] + last;
dfcm[dhash] = u - last;
dhash = ((dhash<<2) XOR ((u-last)>>40)) MOD M;
last = u.
```

FPC calculates differences by bitwise exclusive-OR (XOR) between the true value and the predicted values obtained by FCM and DFCM and selects better predicted value, which presents longer leading zero bytes in the difference. Then, FPC outputs a bit expressing which prediction is adopted, a 3-bit code which means the number of the leading zero bytes (NLZ), and the residual non-zero bytes for a FP number. For example, in Fig. 2, there are 19-bit-length leading zero bits in the better difference, and thus the leading 2-byte zeros are encoded as '010' while the remaining 48 bits including the odd 3-bit-length 0s are output as residuals. In this example, a 64-bit FP number is thus compressed into 52-bit expression.

The compression methods introduced above are basically sequential. A few techniques parallelizing these methods have been proposed [10], [18]. Most of these parallelized methods divide a whole input stream into several fragments and perform the original sequential algorithm for each fragment in parallel. However, because predictors used in such techniques, e.g., FCM and DFCM, depend more or less on all previous inputs, dividing the input stream some-

Fig. 2 Block diagram of FPC.

times diminishes prediction efficiency as mentioned in [10]. That is, we have a trade-off problem in parallel FP data compression between performance improvements by parallel effects and degradation of compression ratio resulting from lowered prediction efficiency. It is desired for massively parallel computers, predictors are free of untoward effects of the partitioned inputs, especially for GPGPU computing.

Therefore, in this paper, we examine a simple predictor based on extrapolation algorithms, which requires only a few FP numbers around the value to be predicted in an input stream, following the previous work using arithmetic predictors [3], [4], [12]. In these techniques, the Lagrange polynomials are used for interpolating or extrapolating a sequence of FP data. A data sequence resulted from a numerical simulation is expected to be smooth, i.e., substantial correlation can be often found between adjacent values [12]. Thus, such data can potentially be well compressed with the arithmetic predictor based compression techniques.

GFC, another derivative of FPC, is originally designed to work on GPUs [11]. It simplifies the predictor of FPC in order to break the dependencies on previous data. GFC uses neither FCM nor DFCM, instead, it use only a previous value to predict. GFC's prediction algorithm is similar to the simplex version of our algorithm presented in the next section. GFC trades off compression ratio against compression throughput on GPUs.

Meanwhile we measured hit rates in bits of the hybrid predictor of FPC for concrete FP data sets of numerical simulations presented in [9] as a preliminary step. As shown in Fig. 3, we found that the bitwise average hit rates are higher for higher bits, however, the rates rapidly decreased for the bits positioned lower than about the 12th bit, which is the most significant bit of mantissa. This means that there are relatively-predictable and hard-to-predict portions in a FP number and the boundary between them is not aligned in byte. Thus, the compression method of FPC, which counts the number of zeros in bytes, cannot take full advantage of the prediction. To improve the compression efficiency for

Fig. 3 Hit rates of predictors of FPC.

such leading zero bytes, we also present a recombination algorithm of several FP numbers in 4. Of course, this algorithm is also designed to be efficiently performed in massively parallel computers.

3. Parallel Predictors

In this section, we propose a fully parallelized predictor with a simple algorithm. Our parallel predictor derives a FP number in an input stream from only h preceding numbers in the stream. It is expected that a resulting data stream from a numerical simulation varies smoothly, and thus we simply predict a value due to extrapolation with the Newton polynomial [19], i.e., the predicted value for the i^{th} value f_i in the input stream $f_1, f_2, \dots, f_N$ is given by $n_h(i)$, where $n_h(x)$ is the h^{th} -order Newton polynomial [19]:

$$n_h(x) = \sum_{j=0}^h [f_x, f_{x-1}, \dots, f_{x-j-1}] \cdot \prod_{k=0}^{j-1} (x - k), \quad (1)$$

where $[f_x, f_{x-1}, \dots, f_{x-j-1}]$ are the backward divided differences defined by

$$[f_x] = [f_x], \quad x \in \{0, 1, \dots, N\}, \quad (2)$$

$$[f_x, \dots, f_{x-k}] = \frac{[f_x, \dots, f_{x-k+1}] - [f_{x-1}, \dots, f_{x-k}]}{k - 1}. \quad (3)$$

Here, we assume that the values in the input stream are placed at regular intervals in transverse. Concretely speaking, the predicted values for f_i are given by the following equations with $h = 0, 1, 2, 3, 4$ and 5:

$$n_0(i) = f_{i-1}, \quad (4)$$

$$n_1(i) = -f_{i-2} + 2f_{i-1}, \quad (5)$$

$$n_2(i) = f_{i-3} - 3f_{i-2} + 3f_{i-1}, \quad (6)$$

$$n_3(i) = -f_{i-4} + 4f_{i-3} - 6f_{i-2} + 4f_{i-1}, \quad (7)$$

$$n_4(i) = f_{i-5} - 5f_{i-4} + 10f_{i-3} - 10f_{i-2} + 5f_{i-1}, \quad (8)$$

$$n_5(i) = -f_{i-6} + 6f_{i-5} - 15f_{i-4} + 20f_{i-3} - 15f_{i-2}$$

Table 1 Average number of leading zero-bytes.

Predictor	brain	comet	control	plasma	EM
FCM (9)	1.35	1.21	0.55	2.28	1.69
FCM (15)	1.47	1.20	0.52	2.06	1.99
FCM (25)	1.38	1.01	0.25	4.82	2.72
DFCM (9)	1.25	1.38	0.69	0.46	2.00
DFCM (15)	1.24	1.36	0.66	4.73	2.12
DFCM (25)	1.37	1.47	0.76	7.96	2.49
GFC	1.17	1.19	0.61	1.25	0.56
n_0	1.38	1.52	0.77	0.49	2.11
n_1	1.36	1.46	0.85	0.25	2.22
n_2	1.22	1.48	0.88	0.25	2.40
n_3	1.07	1.41	0.86	0.25	2.48
n_4	0.94	1.36	0.82	0.25	2.43
n_5	0.83	1.27	0.78	0.25	2.37

$$+ 6f_{i-1}. \quad (9)$$

Especially, $n_0(i)$ simply returns the previous value f_{i-1} .

Numerous processing elements (PEs) calculate the prediction equation above. For example when we have W PEs in a system, each PE performs the algorithm below:

W : the number of processing elements
 N : the number of FP numbers in an input stream
 h : the given parameter of Newton polynomials
 x : the ID of this processing element ($x \in [1, N]$)
 $i \leftarrow x$
while $i \leq N$ **do**
 $pred_i = n_h(i)$
 $i \leftarrow i + W$
end while.

In some massively parallel systems like GPUs, concurrent read accesses to adjacent small memory regions requested by several PEs are automatically coalesced into a single read operation for a large memory block and once the memory block is read, it is saved in fast cache memories [20]. Therefore, PEs which are close to each other can efficiently obtain the input values necessary for prediction with one or a few read accesses.

For compression algorithms mentioned in the previous section, it prefers that a FP number and the corresponding predicted value match at higher rate in more significant bits. For example, FPC encodes the number of succeeding zero bytes counting from the sign bit [9]; in Isenburg's technique, a FP number is compressed efficiently when the true and the predicted values have the same sign and the difference between their exponent is within 3 [5]. Thus we in this paper evaluate predictors by counting number of leading zero bytes (NLZ) in the difference. Table 1 shows the average NLZ for FCM, DFCM, which are the predictors used in FPC, and the proposed Newton-polynomial-based predictors. The parenthetic values written with FCM and DFCM are size parameters for hash tables mentioned in the previous section, that is for example, "FCM (9)" has a hash table with $M = 2^9$ entries. Larger hash table usually presents better compression ratio in both FCM and DFCM [9]. "brain", "comet", "control", and "plasma" are the names of data sets

Table 2 First-order entropies of data sets in Table 1.

	brain	comet	control	plasma	EM
size (MB)	141.8	107.3	159.5	35.09	120.0
1 st -order entropy	23.97	22.04	24.14	13.65	23.84

Table 3 Average number of leading zero-bytes of hybrid predictors.

Predictors	brain	comet	control	plasma	EM
FCM&DFCM (9)	1.46 (0.90)	1.52 (0.70)	0.88 (0.69)	2.37 (0.91)	2.11 (0.59)
FCM&DFCM (15)	1.57 (0.91)	1.53 (0.68)	0.86 (0.68)	5.52 (0.91)	2.40 (0.65)
FCM&DFCM (25)	1.63 (0.76)	1.58 (0.47)	0.81 (0.46)	7.97 (0.49)	2.89 (0.88)
n_0 & n_2	1.46 (0.92)	1.63 (0.88)	1.04 (0.77)	0.49 (1.00)	2.43 (0.68)

resulted from scientific applications presented in [9]; “EM” is a set of FP numbers resulted from a Finite-Difference Time-Domain (FDTD) based simulator for electromagnetic fields [21], [22] implemented by one of the authors [23]. The first-order entropies of these data sets are given in Table 2.

In Table 1, the best value for each data set is indicated by boldface. As shown in this table, we can see that the best predictor differs according to data set. Therefore, it is expected that we can obtain a better predictor suitable for wide variety of data sets by combining a couple of our predictors with different h as FPC adopts a hybrid predictor of FCM and DFCM. Table 3 shows the average NLZ resulted from hybrid predictors. We can see that combinatorial use of two polynomial-base predictors, “ n_0 & n_2 ” always shows better predictability, comparing Tables 1 and 3. Although the Newton-polynomial-base predictors are simple and not new, it is notable that a hybrid predictor of such simple polynomials can be a close competitor against the hysteretic predictors. Moreover, we have little computational overhead for the combinatorial use when we use n_0 , which just gives the previous value as shown in Eq. (4). It is expected that the overhead for comparing NLZs obtained from the two predictors in GPUs is relatively small, comparing with floating-point operations.

The hybrid predictor of FCM and DFCM whose parameter is set to 25 shows the best prediction for three data sets in Table 3, however, the hash tables used in FCM and DFCM totally require about 537 M bytes for each processing element. Our hybrid predictor wins in two data sets; it has much the same effect as FPC (9) for “brain” and FPC (15) for “EM”, respectively. “plasma” is a vibrating data set where signs of succeeding two values are reversed very frequently. Our extrapolation-based predictor is very weak in such data sets.

The parenthesis values located in the lower row of each predictor in Table 3 denote the adoption rates of the predictions generated by FCM or n_0 predictors. We can say that a hybrid predictor achieving long NLZ and having the adoption rate of around 0.5 has the right match. From this viewpoint, the combination of n_0 and n_2 is not good enough in the adoption ratio, comparing to the combinations of

FCM and DFCM. Searching a better parallel prediction algorithm suitable for combination use with the extrapolation-based algorithm is one of our future work.

While we evaluate only prediction effectivity in terms of NLZ in this section, compression ratios and throughputs for these data set will be also presented in 5.

4. Recombination Algorithm

Since predictabilities for lower bits of a FP number decrease rapidly as illustrated in Fig. 3, predicting and compressing a whole FP number with a single algorithm as that used in many conventional techniques is not always efficient. That is, by calculating bitwise difference between true and predicted FP values we can obtain zeros for only a few significant bits of the difference, and there remains unpredictable lengthy chunks in lower bits. Because the compression algorithm of FPC counts NLZ of the difference, FPC can compress only the first byte (8 bits) even when it perfectly predicts the sign bit and the exponent field (12 bits) if it cannot hit the 4 highest bits of the fraction field.

In contrast, the proposed technique compresses FP numbers with two different algorithms. We compress the predictable part of FP numbers by NLZ-based algorithm like FPC. The predictable part in a FP number is very limited: we have only about 12 or so predictable bits per FP number as shown in Fig. 3. It is hard to achieve efficient compression of each FP number which has such small predictable portions by means of NLZ-based methods. Thus in this paper, we combine the significant 16-bit portions of four FP numbers into a 64-bit-length block in order to improve encoding efficiency. We count NLZ of the combined blocks; on the other hand, the remaining 48-bit unpredictable chunk of each FP number is compressed in a different way described later.

Moreover, because the higher the bit position of a FP number is, the more precisely we can predict its value, we interleave bytes or half bytes of the combined block in the order of the indices shown in Fig. 4 in order to lengthen expected NLZ. We have N doubles, $f_1, f_2, \dots, f_N$, to be compressed. We call the interleaved combined block of four differences $d_i, d_{i+1}, d_{i+2}, d_{i+3}$, which are corresponding to the i^{th} to $(i+3)^{\text{th}}$ inputs $f_i, f_{i+1}, f_{i+2}, f_{i+3}$, respectively, as the k^{th} recombinatorial block L_k where $1 \leq k \leq \lceil N/4 \rceil$ and $f_j = 0$ for $j > N$; also, we call $r_i, r_{i+1}, \dots, r_{i+3}$, which are the lower 48 bits of $d_i, \dots, d_{i+3}$ respectively, as residuals. Note that the lower 16 bits of L_k are originated from the highest 4 bits of the fraction part of four FP numbers. These bits are aligned in the lowest portion of L_k because they are relatively difficult to predicate accurately. We count NLZ of L_k, Z_k , and output a 4-bit code expressing Z_k ($0 \leq Z_k \leq 8$). The remaining non-zero bytes of L_k are simply output.

The residuals are hard to predict and thus it seems to have highly random bit patterns to compress with NLZ-based technique or entropy coding. On the other hand, it is expected that in numerical simulations, a same value appears repeatedly in some cases, for example when heat di-

Fig. 4 Recombination of four FP numbers.

Fig. 5 Example of compression of the residuals with Zmap.

vergence converges to a constant very low value at points far from a heat source. In such cases, we can predict the values relatively easily, and the difference can be often zero or a small value. Because of these factors, we compress the residuals whose values equal to zero. We encode the distances of two successive zero-valued residuals in a smaller size than 8 bytes; meanwhile nonzero residuals are simply output without change. Although we cannot expect a substantial contribution in compression ratio for this algorithm, we have only a little penalty in compression ratio even when we have no zero-valued residuals.

Figure 5 illustrates the algorithm for compressing the residuals. In this example, we have 4 residuals, r_1 , r_2 , r_3 , and r_4 , where r_2 and r_3 are zero. In addition, we insert a virtual residual $r_0 (= 0)$ before r_1 for our algorithm. For illustrative

Fig. 6 Example construction of Zmap in a GPGPU implementation.

purposes, we first describe the compression algorithm in sequential style. We maintain a counter p , which memorizes the position of the previous zero-valued residual r_p . We repeatedly search zero-valued residual $r_i (= 0)$ and record only the distance between the two successive zero-valued residuals, $(i - p)$. The algorithm is described in a pseudo code below:

```

p ← 0
for i = 1 to N do
  if r_i = 0 then
    record the distance (i - p)
    p ← i
  else
    output r_i without change
  end if
end for,

```

where $N = 4$ in this example. We call the block describing the distances of pairs of zero-valued residuals as a Zmap. Then, we have two entries in the Zmap, i.e., $(\{r_0, r_2\}, \{r_2, r_3\}) = (2, 1)$. Non-zero residuals, r_1 and r_4 are output without any changes following to the Zmap. A special sequence “0x00...00”, whose length is the same as the encoded size of a distance, is output at the end of the Zmap in order to distinguish non-zero residuals from the Zmap.

In a GPGPU implementation, we can construct the Zmap as illustrated in Fig. 6. We need to output positions of only zero-valued residuals in Zmap. The entry in Zmap to which each thread writes is determined dynamically according to input data and thus is not known in advance. In parallel processing, it is usually addressed by using “prefix sum” [24] algorithm. That is, firstly we make up a bitmap bm whose elements are set as follows:

$$bm[i] = \begin{cases} 1 & (r_i = 0) \\ 0 & (r_i \neq 0) \end{cases} \quad (i = 1, 2, \dots, n). \quad (10)$$

Then, we sum up the bitmap with the prefix sum algorithm. We next compact the prefix-summed array ps into an array cmp by

```

if bm[i] ≠ 0 then

```

```

    cmp[ps[i]] = i
end if,

```

which is performed in parallel. Also, we assume a virtual $cmp[0](= 0)$ for the next step. These three steps are sometimes called as parallel “compact” algorithm [25]. Finally, by differentiating cmp , we obtain a Zmap.

As a whole, the compressed output is organized in a header and four body blocks. The header includes the number of input FP numbers N and the total size of the bodies in bytes. The body blocks are shown in the lower half of Fig. 4. The first block has fixed $(8 \times \lceil N/4 \rceil)$ -bit length: this block consists of N records, each record has 4 bits expressing the selections of hybrid predictor $s_i, \dots, s_{i+3}$ and 4-bit NLZ code Z_k . The second block has a variable length, containing the non-zero bytes of the recombinatorial blocks $L_k (k = 1, 2, \dots, \lceil N/4 \rceil)$. The length of the second block can be known by decoders by analyzing the first block which includes NLZs of the blocks. Finally, the third and fourth blocks are composed as shown in Fig. 5.

5. Numerical Examples and Discussions

In this section, we present some numerical examples of compression ratio and throughput of the proposed technique running on both a CPU and a GPU and compare them with those of some conventional parallelized compression techniques. We implemented the prediction and recombination algorithms presented in the previous sections. We use C++ language and GNU g++ 4.4.3 compiler for the implementation. We also use CUDA 4.0 for the GPU implementation, which is a C-like programming environments for GPGPU presented by NVIDIA. Also, we obtained the source codes of FPC, pFPC and GFC from the authors’ web site [26] and modified them in order to add some time-measuring features. These programs were run on a PC having a 8-core Intel Core i7 processor and an NVIDIA Tesla C2070 GPU installed in the PC. Each cores of the CPU runs at 2.8 GHz; the PC has 24 Gbytes memory. The GPU has 448 PEs, each runs at 1.15 GHz. The PC runs Ubuntu Linux 10.04 LTS.

Table 4 shows compression ratios of some conventional techniques and our technique for the data sets given in Table 2. The compression ratio is obtained by dividing the size of the compressed output by the original file size. The best compression ratio for each data set is indicated by boldface. “gzip” and “bzip2” are general-purpose compression programs, not specialized for compressing FP data. As mentioned in 2, FPC has a parameter which determines the size of hash tables used in its predictors. Both “gzip” and “bzip2” also have a parameter determining trade-off between compression ratio and throughput; we set the parameter so that they have the best compression ration (“-9” option). It can be seen that we can improve compression ratio of FPC by increasing this parameter with using more memory, e.g., it needs more than 537 M bytes for FPC (25). On the other hand, the proposed prediction algorithm, in which we use n_0 and n_2 , surpasses FPC for the data sets except for “plasma”. It can be said that the proposed technique can

Table 4 Compression ratios of several prediction and compression techniques.

	brain	comet	control	plasma	EM
gzip	0.9395	0.8607	0.9456	0.6220	0.9572
bzip2	0.9592	0.8524	0.9719	0.1727	0.9836
FPC (9)	0.8806	0.8730	0.9522	0.7660	0.9406
FPC (15)	0.8670	0.8716	0.9556	0.3721	0.9225
FPC (25)	0.8593	0.8650	0.9609	0.0665	0.8895
pFPC (9)	0.8838	0.8739	0.9561	0.7878	0.9416
GFC	0.9165	0.9137	0.9869	0.9064	0.9945
proposed	0.8411	0.8443	0.9197	0.9710	0.8728

Table 5 Impacts of the interleaving and Zmap techniques.

intrlv.	Zmap	brain	comet	control	plasma	EM
-	-	0.9260	0.9679	0.9865	1.0160	0.8728
✓	-	0.8411	0.8811	0.9251	0.9710	0.8728
-	✓	0.9260	0.9311	0.9452	1.0160	0.8728
✓	✓	0.8411	0.8443	0.9197	0.9710	0.8728

compete with FPC.

In the previous section, we introduced two key techniques as parts of the recombination technique, that are, interleaving the combined block and Zmap. Table 5 shows the effectiveness of the two key techniques on compression ratio. Two columns labeled “intrlv.” and “Zmap” in Table 5 denote enabling/disabling the two techniques, respectively. For most data sets except for “EM”, we can improve the compression ratio with the interleaving technique. Zmap plays a subservient role for enhancing compression ratio. For “brain”, “plasma”, and “EM”, Zmap has no impact: that is, in these datasets, we cannot exactly predict the values at all. On the other hand, Zmap greatly improves the compression ratio for “comet”, which originally includes numerous consecutive 0s.

Next we measured compression throughputs of a couple of parallelized compression techniques running on a CPU. Similarly to pFPC, the CPU implementation of the proposed technique simply divides the input stream into fragments and independently performs the prediction and the recombination algorithms using a thread for each fragment; the number of fragments is equal to the number of used threads. The algorithms are performed sequentially in each thread. FPC, pFPC are compiled by gcc with “-O3” option as instructed in [26]; our implementation is compiled by g++ with “-O2” option.

Figure 7 shows the harmonic mean throughputs in giga-bytes/s (GB/s) varying number of threads used. Compression throughputs are obtained by dividing the original file size by wall-clock time taken for compression operations, excluding file I/O operations. In the figure, “pbzip2” denotes compression throughputs of a parallel implementation of bzip2 [27]. “pbzip2” is configured with a runtime option “-1” so that it has the best throughput.

The CPU used consists of 8 processing cores therefore many techniques show the best performance when we use 8 threads. However, pFPC show lower throughputs with more threads when its hash tables is relatively large. In recent CPUs, uniformly random access to a large mem-

Fig. 7 Compression throughput (GB/s) of parallel compression techniques running on a CPU.

ory block can significantly degrades performance of a program due to frequent cache misses. When data to be compressed is highly random, there is fear that hash tables used in FCM and DFCM are typical examples of such large memory blocks. Especially, “pFPC (25)” sacrifices more than it gains with using more threads. In contrast, the proposed technique always accesses memories linearly, thus it gains its performance with more threads.

We also run GFC and the GPU version of our technique for the same data sets. For the experimental implementation of our technique, we use CUDPP library [25] for the compact algorithm illustrated in Fig. 6. In contrast to the CPU version, we do not divide the input stream. The prediction and recombination algorithms are performed in completely parallel. The GPU throughputs are obtained by dividing the original file size by time taken for GPU processing, not including CPU idle time. The time measurement is performed by inserting `cutStartTimer()` and `cutStopTimer()` function calls, which are provided by NVIDIA, before/after data transfer between CPU and GPU. That is, we exclude time for the CPU-GPU data transfer in evaluating the throughputs. This is because in current GPGPU systems, the data transfer takes several orders of magnitude more time than calculation in a GPU, thus, the data-transfer time hides the time for running the algorithm itself. It is expected that we will be able to utilize some techniques for reducing the data-transfer overhead at an early date [28].

We measure the throughputs of our implementation under two configurations: a) enabling both the interleaving and Zmap techniques, and b) enabling the interleaving and disabling Zmap. Our technique shows throughput at about 4.40 GB/s under configuration a) and 11.10 GB/s under b) on average. This means that the “compact” algorithm needed for generating Zmap has large overhead. We assume this is because the compact algorithm performs the same calculation iteratively and the parallelism of the algorithm decreases in later iteration [24], [25]. As shown in Table 5, the effectiveness of Zmap in compression ratio differs ac-

cording to datasets, thus configuration b) may be sometimes a good choice. GFC outperforms our technique even when we do not use Zmap: it has about 16.29 GB/s. This is partly because our technique needs FP operations for prediction in contrast to GFC’s simple prediction algorithm, in which no mathematical operations are performed. Considering that FP operations in NVIDIA GPUs takes 8 times as lengthy time as integral operations, it is expected that we can improve the throughput of our technique much more by predicting values without FP operations, for example, by converting the FP extrapolating algorithms to fixed-point ones.

Finally, we summarize compression ratio and throughput of several parallel compression techniques examined here:

- pbzip2, one of parallel implementations of conventional compression techniques, has poor throughput (around 20 Mbps).
- pFPC sometimes has higher compression ratio comparing to conventional techniques, such as gzip and bzip2. It is much faster (up to about 1.8 GB/s) than pbzip2.
- The compression algorithm of GFC is quite simple, thus it is the fastest (over 16 GB/s) but poor in compression ratio.
- The proposed technique achieves comparable compression ratio to FPC and pFPC except for “plasma” data set.
- The GPU version of our technique has good throughput at about 4.4 GB/s. By disabling Zmap, we can achieve more throughput. For some datasets, it may be a good choice.

6. Concluding Remarks

In this paper, we propose a highly-parallelizable lossless compression technique for double data resulting from numerical simulations based on the Newton polynomial and a recombination technique in which we combine and interleave bytes or half bytes of several FP numbers in order to improve prediction accuracy. We also implement both CPU and GPU versions of the proposed technique and measure their compression ratio and throughput for several data sets resulting from numerical simulations. Our technique shows comparable compression ratio to FPC and its parallelized version pFPC running on CPUs at higher throughput. Moreover, by using GPU we can improve the compression throughput of our technique.

Our extrapolation-based predictor has a disadvantage in predicting vibrating data sets like “plasma”. We need further study for developing another parallel prediction algorithm which is intended for such vibrating data and suitable for combination use with the predictor presented in this paper.

From a viewpoint of implementation on GPUs, our predictor needs FP operations, which are usually more costly than integer operations, e.g., in NVIDIA GPUs used in this paper, a FP operation takes 8 times as long time as an inte-

ger one. It is expected that predicting the values with only integer operations can could improve compression throughput. We will in the future address integer-based predictors with keeping prediction accuracy. Using fixed-point approximation is one of the promising way to develop such integer predictors.

References

- [1] S. Laosooksathit, N. Naksinehaboon, C. Leangsuksan, A. Dhungana, C. Chandler, K. Chanchio, and A. Farbin, "Lightweight checkpoint mechanism and modeling in GPGPU environment," Proc. 4th Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2010), 2010.
- [2] A. Nukada, H. Takizawa, and S. Matsuoka, "NVCR: A transparent checkpoint-restart library for NVIDIA CUDA," Proc. 2011 IEEE Int'l Parallel Distrib. Processing Symp., pp.104–113, May 2011.
- [3] K. Ktaira, K. Sano, and S. Yamamoto, "FPGA-based lossless compressors of floating-point data streams to enhance memory bandwidth," Proc. 2010 IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors, pp.246–253, July 2010.
- [4] K. Sano, K. Ktaira, and S. Yamamoto, "Segment-parallel predictor for FPGA-based hardware compressor and decompressor of floating-point data streams to enhance memory I/O bandwidth," Proc. 2010 Data Compression Conference, pp.416–425, March 2010.
- [5] M. Isenburg, P. Lindstrom, and J. Snoeyink, "Lossless compression of predicted floating-point geometry," *Comput.-Aided Des.*, vol.37, no.8, pp.869–877, July 2005.
- [6] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Trans. Vis. Comput. Graphics*, vol.12, no.5, pp.1245–1250, 2006.
- [7] L. Ibarria, P. Lindstrom, J. Poirignac, and A. Szymezak, "Out-of-core compression and decompression of large n -dimensional scalar fields," *Comput. Graph. Forum*, vol.22, no.3, pp.343–348, Sept. 2003.
- [8] N. Harada, K. Moriya, H. Sekikawa, K. Siroyanagi, and Y. Kanemoto, "Lossless coding of IEEE 754 floating-point signals for ISO/IEC MPEG-4 Audio Lossless Coding (ALS)," *IEICE Trans. Commun. (Japanese Edition)*, vol.J89-B, no.2, pp.204–213, Feb. 2006.
- [9] M. Burtcher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Trans. Comput.*, vol.58, no.1, pp.18–31, Jan. 2009.
- [10] M. Burtcher and P. Ratanaworabhan, "pFPC: A parallel compressor for floating-point data," Proc. 2009 Data Compression Conference, pp.43–52, 2009.
- [11] M.A. O'Neil and M. Burtcher, "Floating-point data compression at 75 Gb/s on a GPU," Proc. GPGPU'11, March 2011.
- [12] V. Engelson, D. Fritzson, and P. Fritzson, "Lossless compression of high-volume numerical data from simulations," Proc. '00 Data Compression Conference, p.574, March 2000.
- [13] IEEE 754-2008: Standard for Binary Floating-Point Arithmetic, IEEE, Aug. 2008.
- [14] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol.30, no.6, pp.520–540, June 1987.
- [15] G.N.N. Martin, "Range encoding: An algorithm for removing redundancy from a digitised message," Proc. IERE Video & Data Recording Conf., 1979.
- [16] Y. Sazeides and J.E. Smith, "The predictability of data values," Proc. IEEE/ACM 30th Int'l Symp. Microarchitecture (MICRO'97), pp.248–258, Dec. 1997.
- [17] B. Goeman, H. Vandierendonck, and K. Bosschere, "Differential FCM: Increasing value prediction accuracy by improving table usage efficiency," Proc. 7th Int'l Symp. High Performance Computer Architecture (HPCA'01), pp.207–216, Jan. 2001.
- [18] A. Balevic, L. Rockstroh, M. Wroblewski, and S. Simon, "Using arithmetic coding for reduction of resulting simulation data size on massively parallel GPGPUs," Proc. 15th PVM/MPI, pp.295–302, Sept. 2008.
- [19] F.B. Hildebrand, *Introduction to Numerical Analysis*, 2nd ed., Dover Publications, June 1987.
- [20] NVIDIA Corporation, *NVIDIA CUDA Best Practices Guide*, 2010.
- [21] A. Taflove and S.C. Hagness, *Computational electrodynamics: The finite-difference time-domain method*, 3rd ed., Artech House, June 2005.
- [22] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU2), pp.79–84, 2009.
- [23] T. Yamaguchi, "Finite-difference time-domain analysis of hemiteardrop-shaped near-field optical probe," *Electron. Lett.*, vol.44, no.4, pp.310–312, 2008.
- [24] M. Harris, *Parallel Prefix Sum (Scan) with CUDA*, NVIDIA, April 2007.
- [25] "CUDPP: CUDA Data-Parallel Primitives Library," available at http://cudpp.googlecode.com/svn/trunk/doc/CUDPP_slides.pdf
- [26] source codes of FPC, pFPC, GFC are available at <http://www.cs.txstate.edu/~mb92/>
- [27] J. Gilchrist, "Parallel BZIP2 (PBZIP2) data compression software," available at <http://compression.ca/pbzip2/>
- [28] S. Yan, X. Zhou, Y. Gao, H. Chen, G. Wu, S. Luo, and B. Saha, "Optimizing a shared virtual memory system for a heterogeneous CPU-accelerator platform," *ACM SIGOPS Operating Systems Review*, vol.45, no.1, pp.92–100, Jan. 2011.

Mamoru Ohara received his B.E., a M.E., and a Ph.D. from Tokyo Metropolitan University in 2001, 2003, and 2006, respectively. He is now in Tokyo Metropolitan Industrial Technology Research Institute. His research area includes dependable computing, parallel and distributed systems, and embedded systems. He is a member of IEEE.

Takashi Yamaguchi received the B.E., M.E., and Ph.D. degrees in Electrical Engineering from the College of Science and Technology, Nihon University, Japan, in 2003, 2005, and 2008, respectively. He is currently with the Tokyo Metropolitan Industrial Technology Research Institute. His research interests include numerical simulations of electromagnetic waves, optical near-fields, and optical waveguides.