2007－AL－111 (4)
2007／3／9

DNA 計算による浮動小数点演算アルゴリズム

徳丸 雄一郎　藤原 暁宏
九州工業大学 情報工学部 電子情報工学科

概要： 本研究では, DNA 計算を用いて浮動小数点数の演算を行うアルゴリズムを提案する. まずはじめに, DNA による浮動小数点表現を定義する. 本研究で取り扱う浮動小数点数は符号部, 指数部, 仮数部から構成され, 符号部は 1 ビットの 2 進数であり, また, 指数部と仮数部は, それぞれ q ビットと m ビットの 2 進数であるものとする. 次に, 浮動小数点数対に対して加算を行う 2 つのアルゴリズムを提案する. 1 つ目のアルゴリズムは $O(m^2)$ 種類の DNA を用いることにより $O(\log m)$ ステップで実行可能であり, 2 つ目のアルゴリズムは $O(m^2 2^m)$ 種類の DNA を用いることにより $O(1)$ ステップで実行可能である. また, $O(n)$ 個の対に対する浮動小数点数の加算を行うアルゴリズムも提案する. このアルゴリズムは, $O(nm^2 2^m)$ 種類の DNA を用いることにより, $O(1)$ ステップで実行可能である. また最後に, 浮動小数点数の乗算を行うアルゴリズムを提案する. このアルゴリズムは $O(m^2)$ 種類の DNA を用いることにより $O(\log m)$ ステップで実行可能である.

Procedures for floating point arithmetic operations with DNA molecules

Yuichiro Tokumaru, Akihiro Fujiwara
Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology

Abstract: In this paper, we consider procedures for floating point arithmetic operations with DNA molecules. We first propose data structure for representing floating point numbers with DNA molecules. The floating point number consists of a sign bit, an exponent and a mantissa. We assume that the exponent and the mantissa are binary numbers of q and m bits, respectively. We next propose two procedures for an addition of floating point numbers. The first procedure executes an addition in $O(\log m)$ steps using $O(m^2)$ DNA strands, and the second procedure executes an addition in $O(1)$ steps using $O(m^2 2^m)$ DNA strands. We also propose a procedure for additions of $O(n)$ pairs of two floating point numbers. The procedure executes $O(n)$ additions simultaneously in $O(1)$ steps using $O(nm^2 2^m)$ DNA strands. We finally propose a procedure for a multiplication of a pair of two floating point numbers. The procedure executes a multiplication in $O(\log m)$ steps using $O(m^2)$ DNA strands.

1 Introduction

In recent works for high performance computing, computation with DNA molecules, that is, DNA computing, has had considerable attention as one of non-silicon based computing. DNA molecules have two important features, which are Watson-Crick complementarity and massive parallelism. Using the features, we can solve NP optimization problems, which usually need exponential time on a silicon based computer, in a polynomial number of steps with DNA molecules. As the first work for DNA computing, Adleman [1] presented an idea of solving the Hamiltonian path problem of size n in $O(n)$ steps using DNA molecules. His idea was successfully tested in a lab experiment for a small graph. There are a number of other works with DNA molecules for combinatorial NP optimization problems [2, 3, 12, 13, 20].

However, procedures for primitive operations, such as logic or arithmetic operations, are needed to apply DNA computing on a wide range of problems. A number of procedures have been proposed for the primitive operations with DNA molecules [4, 5, 9, 10, 11, 15, 18]. Recently, Santis et. al. [18] have proposed procedures for executing primitive operations for floating point numbers. They assume that each floating point number consists of a sign bit, an exponent, and a mantissa, and that the exponent and the mantissa are binary numbers of q and m bits, respectively. Using the DNA encoding, they proposed two procedures for an addition and a multiplication of two floating point numbers. The procedure for an addition runs in $O(q + \log m)$ steps, and the procedure for a multiplication runs in $O((\log m)^2)$ steps.

In this paper, we consider procedures for floating point arithmetic operations with DNA molecules. We first propose data structure for representing floating point numbers with DNA molecules. We also assume that the exponent and the mantissa are binary numbers of q and m bits, respectively.

We next propose two procedures for an addition of two floating point numbers using the above data structure. In the first procedure for addition, we equalize the exponents of two floating point numbers, and normalize the result of the addition. Then, we shift the mantissa so as to obtain the appropriate value. To realize the procedure with DNA molecules, we first create all candidates for the appropriately shifted value, and then, extract the appropriate value from the candidates. The first procedure runs in $O(\log m)$ steps using $O(m^2)$ DNA strands.

In the second procedure for addition, we first create the candidates for all possible input values.

The creation enables us to execute additions of any pair of floating point numbers with the candidates repeatedly. The second procedure runs in $O(1)$ steps using $O(m^2 2^m)$ DNA strands. Both of the procedures work faster than the previous procedure [18].

We also propose a procedure for additions of $O(n)$ pairs of two floating point numbers. The procedure is an expansion of the second procedure, that is, the procedure consists of parallel executions of the second procedure for $O(n)$ pairs. The procedure runs in $O(1)$ steps using $O(nm^2 2^m)$ DNA strands.

We finally propose a procedure for a multiplication of two floating point numbers. The procedure runs in $O(\log m)$ steps using $O(m^2)$ DNA strands. The procedure also works faster than the previous procedure [18].

2 Preliminaries

2.1 Computational model for DNA computing

A number of theoretical or practical computational models have been proposed for DNA computing [2, 5, 10, 11, 12, 15, 16, 17]. A computational model used in this paper is the same model as [5]. We briefly introduce the model in this section.

A *single strand* of DNA is defined as a string of symbols over a finite alphabet Σ. We define the alphabet $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{m-1}, \overline{\sigma_0}, \overline{\sigma_1}, \ldots, \overline{\sigma_{m-1}}\}$. In this alphabet, the symbols σ_i, $\overline{\sigma_i}$ $(0 \le i \le m-1)$ are *complements*. Two single strands can form a *double strand* if and only if the single strands are complements of each other. A double strand with σ_i, $\overline{\sigma_i}$ is denoted by $\begin{bmatrix} \sigma_i \\ \overline{\sigma_i} \end{bmatrix}$.

The single or double strands are stored in *test tubes*. For example , $T_1 = \{\sigma_0\sigma_1, \overline{\sigma_0\sigma_1}\}$ denotes a test tube that includes two kinds of single strands, $\sigma_0\sigma_1$ and $\overline{\sigma_0\sigma_1}$.

Using the DNA strands, the nine manipulations, which are *Merge*, *Copy*, *Detect*, *Separation*, *Selection*, *Cleavage*, *Annealing*, *Denaturaton* and *Empty*, are allowed on the computational model. Since these nine manipulations are implemented with a constant number of biological steps for DNA strands [14], we assume that the complexity of each manipulation is $O(1)$ steps. (See [5] for more details.)

2.2 Representation of binary numbers with DNA molecules

In this section, we describe a representation of n binary numbers of m bits. In the representation, one single strand corresponds to one bit of a binary number. Therefore, we use $O(mn)$ single strands to denote n binary numbers.

We first define the alphabet Σ used in the representation as follows.

$$\begin{aligned}\Sigma = \{&A_0, A_1, \ldots, A_{n-1}, B_0, B_1, \ldots, B_{m-1},\\ &a_S, a_E, b_S, b_E, C_0, C_1, D_0, D_1, 0, 1, \sharp,\\ &\overline{A_0}, \overline{A_1}, \ldots, \overline{A_{n-1}}, \overline{B_0}, \overline{B_1}, \ldots, \overline{B_{m-1}},\\ &\overline{a_S}, \overline{a_E}, \overline{b_S}, \overline{b_E}, \overline{C_0}, \overline{C_1}, \overline{D_0}, \overline{D_1}, \overline{0}, \overline{1}, \overline{\sharp}\}\end{aligned}$$

In the above alphabet, $A_0, A_1, \ldots, A_{n-1}$ denote addresses of binary numbers, and $B_0, B_1, \ldots, B_{m-1}$ denote bit positions, in which each binary number is stored. More precisely, A_i and B_j are defined as follows using $a_S, a_E, b_S, b_E, 0, 1$.

$$\begin{aligned} A_i &= a_S a_{(\log n)-1} a_{(\log n)-2} \cdots a_1 a_0 a_E \\ B_j &= b_S b_{(\log m)-1} b_{(\log m)-2} \cdots b_1 b_0 b_E \end{aligned}$$

In the above description, $a_{(\log n)-1}$, $a_{(\log n)-2}$, $\ldots$, a_1, a_0 are binary numbers such that $i = \sum_{k=0}^{(\log n)-1} a_k \times 2^k$, and $b_{(\log m)-1}$, $b_{(\log m)-2}, \ldots, b_1$, b_0 are binary numbers such that $j = \sum_{k=0}^{(\log m)-1} b_k \times 2^k$, where each a_k and b_k is 0 or 1. In addition, a_S, a_E and b_S, b_E are start and terminal symbols, respectively. We also assume that C_0, C_1 and D_0, D_1 are the specified symbols cut by *Cleavage*. $\sharp$ is a special symbol for *Separation*.

Using the above alphabet, a value of a bit, whose address and bit position are i and j, is represented by a single strand $S_{i,j}$ such that

$$\begin{aligned} S_{i,j} &= D_1 a_S a_{(\log n)-1} \cdots a_1 a_0 a_E \\ &\quad b_S b_{(\log m)-1} \cdots b_1 b_0 b_E C_0 C_1 V_{i,j} D_0 \\ &= D_1 A_i B_j C_0 C_1 V_{i,j} D_0 \end{aligned}$$

, where $V_{i,j} = 0$ if a value of the bit is 0, otherwise $V_{i,j} = 1$.

We call each $S_{i,j}$ a *memory strand*, and use a set of $O(mn)$ different memory strands to denote n binary numbers of m bits.

We also assume that $S_{i,j}(V)$ denote a memory strand whose value is V as follows.

$$\begin{aligned} S_{i,j}(0) &= D_1 A_i B_j C_0 C_1 0 D_0 \\ S_{i,j}(1) &= D_1 A_i B_j C_0 C_1 1 D_0 \end{aligned}$$

2.3 Primitive operations

In this paper, we use five operations as primitive operations. The input and output of the primitive operations are defined in the following.

1. **Logic operations :** This operation executes logic operations defined by a truth table L for the memory strands in the test tube T_{input}. The output is stored in the test tube T_{output}.

   Notation: $Logic(T_{input}, L, T_{output})$

2. **Additions and subtractions :** This operation executes additions or subtractions for pairs of the memory strands in the test tube T_{input}. The pairs of additions or subtractions are defined by a test tube R. The output is stored in the test tube T_{output}.

   Notation: $Addition(T_{input}, R, T_{output})$, $Subtraction(T_{input}, R, T_{output})$

3. **Multiplication :** This operation executes a multiplication of a pair of memory strands, which denote two binary numbers. Two sets of input memory strands are stored in T_{input_x} and T_{input_y}, and the output of a multiplication is stored in the test tube T_{output}.

   Notation:
   $Multiplication(T_{input\_x}, T_{input\_y}, T_{output})$

4. **Address extraction :** This operation creates special single strands that correspond to memory strands in the input test tube. T_{input} is a test tube that contains memory strands, and T_{splint} is another set of special single strands. T_{output} is an output test tube that contains special single strands corresponding to memory strands in T_{input}. Test tubes T_{splint} and T_{output} are defiled below.

$$\begin{aligned} T_{splint} &= \{\overline{A_i^R D_0 D_1 A_i} \mid 0 \leq i \leq n-1\} \\ T_{output} &= \{D_1 A_i^R D_0 \mid S_{i,j} \in T_{input}\} \end{aligned}$$

   Notation:
   $ExtractAddress(T_{input}, T_{splint}, T_{output})$

5. **Memory strand extraction :** This operation extracts memory strands that correspond to the special single strands obtained by $ExtractAddress$. T_{input} is a test tube that contains special single strands obtained by $ExtractAddress$, and T_{splint} is a test tube that contains another set of special single strands. T_{output} is a test tube that initially contains memory strands. At the end of the operation, T_{output} contains the memory strands corresponding to special single strands in T_{input}. Test tubes T_{splint} and T_{output} at the end of the operation are defined below.

$$\begin{aligned} T_{splint} &= \{\overline{A_i^R D_0 D_1 A_i} \mid 0 \leq i \leq n-1\} \\ T_{output} &= \{S_{i,j} \mid \overline{A_i^R D_0 D_1 A_i} \in T_{input}\} \end{aligned}$$

   Notation:
   $ExtractMemory(T_{input}, T_{splint}, T_{output})$

The followings are lemmas obtained for the primitive operations.

Lemma 1 [5] *Logic operations for $O(n)$ pairs of m-bit binary numbers can be executed in $O(1)$ steps using $O(mn)$ DNA strands.* □

Lemma 2 [5] *Additions and subtractions for $O(n)$ pairs of m-bit binary numbers can be executed in $O(1)$ steps using $O(mn)$ DNA strands.*□

Lemma 3 [6] *A multiplication of a pair of m-bit binary numbers can be executed in $O(\log m)$ steps using $O(m^2)$ DNA strands.* □

Lemma 4 [7] *Address extraction can be executed in $O(1)$ steps using $O(1)$ DNA strands.* □

Lemma 5 [7] *Memory strand extraction can be executed in $O(1)$ steps using $O(1)$ DNA strands.* □

3 Representation of floating point numbers

The floating point is the most common representation for real numbers on conventional computers. The most common formula for the floating point is IEEE Standard 754 [8]. In the standard, each floating point number consists of three components: the sign bit s, the exponent e, and the mantissa f. Using these components, a real number is written as follows.

$$(-1)^s \times f \times 2^e$$

In the above expression, s is the binary bit such that $s = 0$ in case that the floating point number is positive, otherwise, $s = 1$.

The exponent e is a binary number of $q+1$ bits, and represents both positive and negative numbers. In the IEEE standard, a bias is added to the actual exponent and stored in e because e is kept to be positive. In other words, the value stored in e is $a + 2^{q-1} - 1$ if the actual exponent is a. For example, in case that $q = 8$, 127 is stored in e if the actual exponent is 0. In addition to the above, we assume that the left most bit of the exponent is a sign bit of the exponent because we execute subtraction between exponents in our procedures.

The mantissa f is a binary number of $2m+1$ bits, and represents precision bits of a real number. We also assume that the leftmost bit of the mantissa is a sign bit of the mantissa because we execute subtraction between mantissas in our procedures. The next m bits represent the integer part, and the final m bits represent the decimal part. We assume that the mantissa is normalized so that $1 \leq f < 2$.

We show an example of the floating point number in case that $q = 8$, $m = 4$, and the real number is -4.

$$s = 1, \quad e = 010000001, \quad f = 00001.0000$$

In this paper, we assume that the following three kinds of memory strands, which are $S_{i,m}$, $E_{i,j}$ and $F_{i,k}$, represent the sign bit, the exponent, and the mantissa of a floating point number. In the following description, the valuable i denotes an address, and the variable m denotes the number of bits of the decimal or integer part of the mantissa.

$$\begin{aligned} S_{i,m} &= D_1 A_i B_m C_0 C_1 V_{i,m} D_0 \\ E_{i,j} &= D_1 A_i B_j C_0 C_1 V_{i,j} D_0 \quad (0 \leq j \leq q) \\ F_{i,k} &= D_1 A_i B_k C_0 C_1 V_{i,k} D_0 \quad (-m \leq k \leq m) \end{aligned}$$

We also assume that s_i, e_i and f_i are binary numbers that denote the sign bit, the exponent and the mantissa, which are stored in address i.

Input and output of the procedures for addition or multiplication are the following test tubes. In the description, x and y are addresses in which the floating point numbers X and Y are stored, respectively. A result of the procedure is stored in the address x.
(Input)

$$\begin{aligned} T_s &= \{S_{i,m} \mid i \in \{x,y\}\} \\ T_e &= \{E_{i,j} \mid i \in \{x,y\},\ 0 \leq j \leq q\} \\ T_f &= \{F_{i,k} \mid i \in \{x,y\},\ -m \leq k \leq m\} \end{aligned}$$

(Output)

$$\begin{aligned} T_{ans\_s} &= \{S_{x,m}\} \\ T_{ans\_e} &= \{E_{x,j} \mid 0 \le j \le q\} \\ T_{ans\_f} &= \{F_{x,k} \mid -m \le k \le m\} \end{aligned}$$

4 Procedures for addition of floating point numbers

In this section, we first propose two procedures for an addition of floating point numbers. The first procedure executes an addition of a pair of floating point numbers in $O(\log m)$ steps using $O(m^2)$ DNA strands, and the second procedure executes the same addition in $O(1)$ steps using $O(m^2 2^m)$ DNA strands. We also proposed a procedure for additions of $O(n)$ pairs of floating point numbers. The procedure runs in $O(1)$ steps using $O(nm^2 2^m)$ DNA strands. Due to space limitation, we describe outline of the procedures only. (See [19] for details of the procedure.)

4.1 The first procedure for addition

We first describe an overview of the procedure, which is called $FloatAdd1$. The basic steps of the procedure $FloatAdd1$ are as follows.

Step 1 : Exchange the values between X and Y if $e_y > e_x$.

Step 2 : Compare the exponents, and exit the procedure if the difference is larger than m.

Step 3 : Shift the mantissa of Y right by an amount equal to the difference between the exponents.

Step 4 : Compute 2's complements of two mantissas of X and Y. Then, add two computed complements, and compute 2's complement of the result.

Step 5 : Normalize the mantissa, and adjust the exponent.

In Step 1, we compute $e_x - e_y$, and exchange the values between X and Y if the result of the subtraction is negative. In Step 2, we subtract $m+1$ from the difference between the exponents, which is computed in Step 1. Then, we output X and exit the procedure if the result of the subtraction is positive.

In Step 3, we first create the candidates, which are all possible right-shifted values of the mantissa f_y. If $f_y = 00001.0000$, the candidates R_w $(0 \le w \le m)$ are shown in Table 1. We next select actual shift amount, which is equal to the difference between the exponents, from all possible shift amounts Z_w $(0 \le w \le m)$. The selection is executed by subtracting the difference between the exponents from all possible shift amounts. The operation is shown in Table 2 in case that the difference is 2. In the table, D denotes the difference. In this case, R_2 is the value shifted right by an amount equal to the difference.

Table 1: An example of candidates

w	R_w
0	00001.0000
1	00000.1000
2	00000.0100
3	00000.0010
4	00000.0001

Table 2: Result of Step 3

Z_w	D	$Z_w - D$	R_w
0	2	-2	00001.0000
1	2	-1	00000.1000
2	2	$\boxed{0}$	$\boxed{00000.0100}$
3	2	1	00000.0010
4	2	2	00000.0001

In Step 4, we compute 2's complements of two mantissas of X and Y because input floating point numbers may be negative numbers. Then, we add two computed complements, and compute 2's complement of the result since the mantissa must be kept positive.

In Step 5, we normalize the result of the addition according to the following three cases.

$$\begin{aligned} Case\,1 &: 2 \le f_x < 4, \text{ that is, } f_x = 0001{*}.{*}{*}{*}{*} \\ Case\,2 &: 1 \le f_x < 2, \text{ that is, } f_x = 00001.{*}{*}{*}{*} \\ Case\,3 &: 0 \le f_x < 1, \text{ that is, } f_x = 00000.{*}{*}{*}{*} \end{aligned}$$

In $Case$ 1, we shift the mantissa right by 1 bit, and add 1 to the exponent. In $Case$ 2, the result of an addition is the normalized number , and we leave the result as it is. In $Case$ 3, we create the candidates L_w $(0 \le w \le m)$, which are all possible left-shifted values of the results. Then, we select the normalized value from the candidates, and subtract the shift amount from the exponent.

We next list test tubes used in the procedure in the following. In the following description, we assume that α is an arbitrary constant.

T_{shift_r1} : The test tube T_{shift_r1} contains the memory strands $R^1_{\alpha+w,k}(0)$ that store the candidates, which are all possible right-shifted values of the mantissa f_y. We also assume that $r^1_{\alpha+w}$ are binary numbers denoted by the above memory strands.

T_{shift_l1} : The test tube T_{shift_l1} contains the memory strands $L^1_{\alpha+w,k}(0)$ that store the candidates, which are all possible left-shifted values of the result of an addition. We also assume that $l^1_{\alpha+w}$ are binary numbers denoted by the above memory strands.

T'_{shift_l1} : The test tube T_{shift_l1} is copied to the test tube T'_{shift_l1}.

T_{val_r1} : The test tube T_{val_r1} contains the memory strands $Z^{r1}_{\alpha+w,j}$ that denote all possible shift amounts for the exponent alignment. We also assume that $z^{r1}_{\alpha+w}$ are binary numbers denoted by the above memory strands.

T_{val_l1} : The test tube T_{val_l1} contains the memory strands $Z^{l1}_{\alpha+w,j}$ that denote all possible shift amounts for the normalization. We also assume that $z^{l1}_{\alpha+w}$ are binary numbers denoted by the above memory strands.

T_{e_sub} : The test tube T_{e_sub} stores the difference between the exponents.

$T_{splint1}$: The test tube $T_{splint1}$ contains the single strands that are used for $ExtractAddres$ and $ExtractMemory$.

T_0 : The test tube T_0 stores the memory strands whose values are 0.

T_1 : The test tube T_1 stores the memory strands whose values are 1.

T_{temp} : The test tube T_{temp} is temporarily used for various operations.

T_{trash} : The test tube T_{trash} is used to discard needless DNA strands.

In addition, we define the memory strands $N_{\beta,j}$, which denote a value $m+1$, for the procedure. $N_{\beta,j}$ is given as follows. (We assume that β is an arbitrary constant, and that n_β is a binary number denoted by $N_{\beta,j}$.)

$$N_{\beta,j} = D_1 A_\beta B_j C_0 C_1 V_{\beta,j} D_0 \quad (0 \le j \le q)$$

We also define the memory strands $O_{\delta,j}$ and $O_{\delta,k}$, which both denote the value 1, for the procedure. $O_{\delta,j}$ and $O_{\delta,k}$ are given as follows. (We assume that δ is an arbitrary constant, and that o^j_δ and o^k_δ are binary numbers denoted by $O_{\delta,j}$ and $O_{\delta,k}$.)

$$\begin{aligned} O_{\delta,j} &= D_1 A_\delta B_j C_0 C_1 V_{\delta,j} D_0 \quad (0 \le j \le q) \\ O_{\delta,k} &= D_1 A_\delta B_k C_0 C_1 V_{\delta,k} D_0 \quad (-m \le k \le m) \end{aligned}$$

We now describe the basic steps of the procedure $FloatAdd1$. In the following description, we assume that $0 \le j \le q$, $-m \le k \le m$, and $0 \le w \le m$.

Procedure *FloatAdd1*

Step 1 Exchange the values between X and Y if $e_y > e_x$.

(1-1) Execute the subtraction $e_x = e_x - e_y$ in the test tube T_e, and store the result in the test tube T_{e_sub}.

(1-2) Extract the sign bit of the result of (1-1) from T_{e_sub} if the value of the bit is 1, and store the extracted memory strands in the test tube T_1 by $Separation$. Then, if any memory strand is detected in T_1 by $Detect$, e_y is larger than e_x. Therefore, exchange the sign bits, the exponents, and the mantissas between X and Y by $Logic$ in the test tubes T_s, T_e, and T_f, respectively.

(1-3) Execute the subtraction $e_x = e_x - e_y$ in T_e again, and store the result in T_{e_sub}.

Step 2 Compare the exponents, and exit the procedure if the difference is larger than m.

(2-1) Copy T_{e_sub} to the test tube T_{temp}, and merge the memory strands $N_{\beta,j}$ that denote a value $m+1$ to T_{temp}. Then, subtract $m+1$ from the difference between the exponents.

(2-2) Extract the sign bit of the result of (2-1) from T_{temp} if the value of the bit is 0, and store the extracted memory strands in the test tube T_0 by $Separation$. Then, if any memory strand is detected in T_0 by $Detect$, the difference between the exponents is larger than m. Therefore, output the memory strands $S_{x,m}$, $E_{x,j}$, and $F_{x,k}$ to the test tubes T_{ans_s}, T_{ans_e}, and T_{ans_f}, respectively. If no memory strand is detected in T_0, continue the procedure.

Step 3 Shift the mantissa of Y right by $e_x - e_y$ bits.

(3-1) Extract $F_{y,k}$ from the test tube T_f, and store the extracted memory strands in the test tube T_{shift_r1} by $Separation$. Then, copy f_y to r^1_α by $Logic$, and return $F_{y,k}$ to T_f by $Separation$.

(3-2) Shift r^1_α right by $1, 2, \ldots, m$ bits in parallel, and store each of the shifted values in $R^1_{\alpha+1,k}, R^1_{\alpha+2,k}, \ldots, R^1_{\alpha+m,k}$ by the following operations using $Logic$.

```
for $(g = 0; g \le \lfloor \log m \rfloor; g++)$
    $V_{\alpha+w+2^g,k-2^g} = V_{\alpha+w,k}$ ;
```

(3-3) Merge the test tube T_{val_r1} that contains $Z^{r1}_{\alpha+w,j}$ to T_{e_sub}, and execute the subtractions $z^{r1}_{\alpha+w} = z^{r1}_{\alpha+w} - (e_x - e_y)$ in T_{e_sub} in parallel. Then, if the result whose address is $\alpha + t$ $(t \in w)$ denotes the value 0, the candidate $R^1_{\alpha+t,k}$ denotes the value shifted right by $e_x - e_y$ bits.

(3-4) Extract the memory strands whose values are 1 from T_{e_sub}, and store the extracted memory strands in T_1 by $Separation$. Then, all bits of the result of (3-3) that denote the value 0 are stored in T_{e_sub}. In other words, the other results of the subtractions in (3-3) have at least one bit whose value is 1. Therefore, extract the result of (3-3) that denotes the value 0 from T_{e_sub}, and store the extracted memory strands in T_{temp}. Then, eliminate the memory strands except $R^1_{\alpha+t,k}$ that denotes the value shifted right by $e_x - e_y$ bits from T_{shift_r1} by $ExtractAddress$ and $ExtractMemory$.

(3-5) Extract $F_{y,k}$ from T_f, and store the extracted memory strands in T_{shift_r1} by $Separation$. Then, copy the right-shifted value to f_y by $Logic$, and return $F_{y,k}$ to T_f by $Separation$.

Step 4 Compute 2's complements of two mantissas of X and Y. Then, add two computed complements, and compute 2's complement of the result.

(4-1) Extract the memory strands whose values are 1 from T_s, and store the extracted memory strands in T_1 by $Separation$. Then, extract the mantissas that do not correspond to the memory strands in T_1 from T_f, and store the extracted mantissas in T_{temp} by $ExtractAddress$ and $ExtractMemory$.

(4-2) Compute NOT for each bit of the mantissas in T_f by $Logic$. Then, merge the memory strands $O_{\delta,k}$ that denote the value 1 to T_f, and add the results of the NOT and the value 1 in parallel.

(4-3) Merge T_{temp} to T_f, and execute the addition $f_x = f_x + f_y$.

(4-4) Extract the sign bit of the result of (4-3) from T_f, and store the extracted memory strands in T_{temp} by $Separation$. Then, copy T_{temp} to T_{ans_s}, and compute 2's complement of the result of (4-3).

Step 5 Normalize the mantissa, and adjust the exponent.

We normalize the result of the addition according to the following three cases.

$$\begin{aligned} Case\,1 &: 2 \leq f_x < 4, \text{that is,} \\ f_x &= 0001{*}.{*}{*}{*}{*} \\ Case\,2 &: 1 \leq f_x < 2, \text{that is,} \\ f_x &= 00001.{*}{*}{*}{*} \\ Case\,3 &: 0 \leq f_x < 1, \text{that is,} \\ f_x &= 00000.{*}{*}{*}{*} \end{aligned}$$

(5-1) Extract $F_{x,1}$ from T_f, and store the extracted memory strands in T_1 by $Separation$ if the value of the bit is 1. Then, if any memory strand is detected in T_1, proceed to (5-2) since the result of an addition corresponds to $Case$ 1, otherwise, proceed to (5-3).

(5-2) First, merge T_1 to T_f, and shift the mantissa right by 1 bit by $Logic$. Then, output the shifted mantissa to T_{ans_f}. Next, merge the memory strands $O_{\delta,j}$ that denote the value 1 to T_e, and execute the addition $e_x = e_x + o^j_\delta$. Finally, output the exponent to T_{ans_e}, and exit the procedure.

(5-3) Extract $F_{x,0}$ from T_f, and store the extracted memory strands in T_1 by $Separation$ if the value of the bit is 1. Then, if any memory strand is detected in T_1, proceed to (5-4) since the result of an addition corresponds to $Case$ 2, otherwise, proceed to (5-5) since the result corresponds to $Case$ 3.

(5-4) In $Case$ 2, since the result of (4-3) is the normalized value, output $E_{x,j}$ and $F_{x,k}$ to T_{ans_e} and T_{ans_f}, respectively, and exit the procedure.

(5-5) In $Case$ 3, since the process of normalization is similar to Step 3, we explain it briefly.

First, merge T_f to the test tube T_{shift_l1}, and copy f_x to l^1_α by $Logic$. Then, return $F_{x,k}$ to T_f by $Separation$.

(5-6) Shift l^1_α left by $1, 2, \ldots, m$ bits in parallel, and store the shifted values in $L^1_{\alpha+1,k}, L^1_{\alpha+2,k}, \ldots, L^1_{\alpha+m,k}$, respectively. These operations are almost the same as (3-2). Then, $L^1_{\alpha+w,k}$ denote the candidates for the normalized value.

(5-7) Copy T_{shift_l1} to the test tube T'_{shift_l1} to preserve the values of the candidates for the normalized value.

(5-8) Merge $O_{\delta,k}$ that denote the value 1 to T'_{shift_l1}, and execute the subtractions $l^1_{\alpha+w} = l^1_{\alpha+w} - o^k_\delta$ in parallel in T'_{shift_l1}. Then, eliminate the results that denote negative values from T'_{shift_l1} by $ExtractAddress$ and $ExtractMemory$.

(5-9) Merge $O_{\delta,k}$ to T'_{shift_l1} again, and subtract 1 from the results of (5-8) in parallel. Then, if $(l^1_{\alpha+t} - o^k_\delta) - o^k_\delta$ $(t \in w)$ is negative, $1 \leq l^1_{\alpha+t} < 2$, that is, $l^1_{\alpha+t}$ is the normalized value.

(5-10) Eliminate the memory strands except $L^1_{\alpha+t,k}$ $(t \in w)$ that denote the normalized value from T_{shift_l1} by $ExtractAddress$ and $ExtractMemory$.

(5-11) Eliminate the memory strands except $Z^{l1}_{\alpha+t,j}$ whose address corresponds to the result of (5-10) from the test tube T_{val_l1} by $ExtractAddress$ and $ExtractMemory$.

(5-12) Merge T_{val_l1} to T_e, and execute the subtraction $e_x = e_x - z^{l1}_{\alpha+w}$. Then, output the result to T_{ans_e}.

(5-13) Merge T_f to T_{shift_l1}, and copy the normalized value to f_x by $Logic$. Finally, output $F_{x,k}$ to T_{ans_f} by $Separation$.

(End of the procedure)

Although we omit details of the procedure due to space limitation, we obtain the following theorem for the above procedure $FloatAdd1$. (See [19] for details of the procedure.)

Theorem 1 *The procedure $FloatAdd1$, which computes an addition of a pair of floating point numbers whose exponents and mantissas are binary numbers of q and m bits respectively, runs in $O(\log m)$ steps using $O(m^2)$ different additional DNA strands.* □

4.2 Other procedures for addition

We obtain two more theorems for addition with DNA molecules. We also omit details of the procedures due to space limitation (See [19] for details of the procedure.)

Theorem 2 *An addition of a pair of floating point numbers, whose exponents and mantissas are binary numbers of q and m bits respectively, can be computed in $O(1)$ steps using $O(m^2 2^m)$ different additional DNA strands.* □

Theorem 3 *Additions of $O(n)$ pairs of floating point numbers, whose exponents and mantissas are binary numbers of q and m bits respectively, can be computed in $O(1)$ steps using $O(nm^2 2^m)$ different additional DNA strands.* □

5 Procedure for multiplication

In this section, we propose a procedure for multiplication of floating point numbers. The procedure, which is called $FloatMultiplication$, executes a multiplication of a pair of floating point numbers in $O(\log m)$ steps using $O(m^2)$ DNA strands.

The procedure for multiplication is simple in comparison with the procedures for addition because we use known algorithms for multiplication [6].

We first show test tubes that are not empty at the beginning of the procedure.

T_b **:** The following test tube T_b contains the memory strands $B_{b,j}$ that denote a bias value $2^{q-1}-1$. (We assume that b is an arbitrary constant.)

$$T_b = \{B_{b,j} \mid 0 \le j \le q\}$$

We now show an overview of the procedure $FloatMultiplication$. In the following description, we assume that $0 \le j \le q$, $-m \le k \le m$.

Procedure *FloatMultiplication*

Step 1 Subtract a bias value $2^{q-1}-1$ from e_y, and add the result to e_x, that is, execute $e_x = e_x + (e_y - b_b)$.

Step 2 Execute the multiplication $f_x = f_x \times f_y$ by $Multiplication$.

Step 3 Normalize the result of a multiplication.

(3-1) Extract $F_{x,1}$ for T_f if the value of the bit is 1, and store the extracted memory strand in T_1 by $Separation$. Then, if any memory strand is detected in T_1 by $Detect$, proceed to (3-2), otherwise, output the sign bit and the mantissa to T_{ans_e} and T_{ans_f}, respectively, and then, proceed to Step 4.

(3-2) Merge T_1 to T_f, and shift the mantissa right by 1 bit by $Logic$. Then, output the mantissa to T_{ans_f}.

(3-3) Merge the memory strands $O_{\delta,j}$ that denote the value 1 to T_e, and add 1 to the exponent. Then, output the exponent to T_{ans_e}.

Step 4 Compute $s_x = s_x \oplus s_y$ in T_s by $Logic$, and output the result to T_{ans_s}.

(End of the procedure)

5.1 Details of the procedure

In this subsection, we describe details of the procedure. We first show the address pairs for additions and a subtraction, the single strands that are necessary for logic operations that we execute in $FloatMultiplication$ in the following.

R : The set of address pairs $R = \{(x,\psi) | \psi \in \{y,\ \delta\}\}$

R_b : The address pair $R_b = \{(y,b)\}$

L_{ex} : The set of single strands L_{ex} is used to compute $s_x \oplus s_y$.

L_{r1} : The set of single strands L_{r1} is used to shift f_x right by 1 bit.

We summarize details of the procedure $FloatMultiplication$ in Figure 1.

We finally consider the complexity of the above procedure. Step 1, Step 3, and Step 4 consist of a constant number of DNA manipulations, respectively. In Step 3, we execute $Multiplication$ that runs in $O(\log m)$ steps using $O(m^2)$ additional DNA strands. Therefore, we obtain the following theorem for the above procedure.

Theorem 4 *The procedure $FloatMultiplication$, which executes a multiplication of a pair of floating point numbers whose exponents and mantissas are binary numbers of q and m bits respectively, runs in $O(\log m)$ steps using $O(m^2)$ different additional DNA strands.* □

6 Conclusions

In this paper, we proposed procedures for floating point arithmetic operations with DNA molecules. We first proposed data structure for representing floating point numbers with DNA molecules. We next proposed two procedures for an addition of

```
**Procedure** $FloatMultiplication(T_s, T_e, T_f, T_{ans\_s}, T_{ans\_e}, T_{ans\_f})$

/* Step 1: $e_x + (e_y - b_b)$ */
$Separation(T_e, \{A_y\}, T_b)$;
$Subtraction(T_b, R_b, T_b)$;
$Merge(T_e, T_b)$;
$Addition(T_e, R, T_e)$;

/* Step 2: Multiplication of the mantissas */
$Separation(T_f, \{A_y\}, T_{fy})$;
$Multiplication(T_f, T_{fy}, T_f)$;

/* Step 3: Normalization */
/* (3-1) */
$Separation(T_f, \{B_1 C_0 C_1 1\}, T_1)$;
if ($Detect(T_1) ==$ "$yes$"){
    /* (3-2) */
    $Merge(T_f, T_1)$; $Logic(T_f, L_{r1}, T_f)$;
    $Merge(T_{ans\_f}, T_f)$;
    /* (3-3) */
    $Merge(T_e, \{O_{\delta,j}\})$; $Addition(T_e, R, T_{ans\_e})$;
}
else { $Merge(T_{ans\_e}, T_e)$; $Merge(T_{ans\_f}, T_f)$; }

/* Step 4: Determination of the sign bit */
$Logic(T_s, L_{ex}, T_s)$;
$Separation(T_s, \{A_x\}, T_{ans\_s})$;
```

Figure 1: A procedure for multiplication.

two floating point numbers. The first procedure runs in $O(\log m)$ steps using $O(m^2)$ DNA strands, and the second procedure runs in $O(1)$ steps using $O(m^2 2^m)$ DNA strands. We also proposed a procedure, which runs in $O(1)$ steps using $O(nm^2 2^m)$ DNA strands, for additions of $O(n)$ pairs of two floating point numbers. We finally proposed a procedure for a multiplication of two floating point numbers. The procedure runs in $O(\log m)$ steps using $O(m^2)$ DNA strands.

Since our results are based on a theoretical model, some defects are involved in the procedures for practical use. However, every DNA manipulation used in the model has been already realized in lab level, and some procedures can be implemented practically. Therefore, we believe that our results play important role in the future DNA computing.

References

[1] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266:1021–1024, 1994.

[2] L. M. Adleman. Computing with DNA. *Scientific American*, 279(2):54–61, 1998.

[3] E. B. Baum and D. Boneh. Running dynamic programming algorithms on a DNA computer. In *Proceedings of the Second Annual Meeting on DNA Based Computers*, 1996.

[4] P. Frisco. Parallel arithmetic with splicing. *Romanian Journal of Information Science and Technology*, 2(3):113–128, 2000.

[5] A. Fujiwara, K. Matsumoto, and W. Chen. Procedures for logic and arithmetic operations with DNA molecules. *International Journal of Foundations of Computer Science*, 15(3), 2004.

[6] H. Fukagawa. Procedures for multiplication and division in DNA computing. Bachelor's thesis, Kyushu Institute of Technology, 2005.

[7] K. Fukumoto. Studies on algorithms for memory strands in DNA computing. Master's thesis, Kyushu Institute of Technology, 2005.

[8] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[9] F. Guarnieri, M. Fliss, and C. Bancroft. Making DNA add. *Science*, 273:220–223, 1996.

[10] V. Gupta, S. Parthasarathy, and M. J. Zaki. Arithmetic and logic operations with DNA. In *Proceedings of the 3rd DIMACS Workshop on DNA Based Computers*, pages 212–220, 1997.

[11] H. Hug and R. Schuler. DNA-based parallel computation of simple arithmetic. In *Proceedings of the 7th International Meeting on DNA Based Computers*, pages 159–166, 2001.

[12] R. J. Liption. DNA solution of hard computational problems. *Science*, 268:542–545, 1995.

[13] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber. DNA solution of the maximal clique problem. *Science*, 278:446–449, 1997.

[14] G. Păun, G. Rozeberg, and A. Salomaa. *DNA computing*. Springer-Verlag, 1998.

[15] Z. F. Qiu and M. Lu. Arithmetic and logic operations for DNA computers. In *Proceedings of the Second IASTED International conference on Parallel and Distributed Computing and Networks*, pages 481–486, 1998.

[16] Z. F. Qiu and M. Lu. Take advantage of the computing power of DNA computers. In *Proceedings of the Third Workshop on Bio-Inspired Solutions to Parallel Processing Problems, IPDPS 2000 Workshops*, pages 570–577, 2000.

[17] J. H. Reif. Parallel biomolecular computation: Models and simulations. *Algorithmica*, 25(2/3):142–175, 1999.

[18] F. D. Santis and G. Iaccarino. A DNA arithmetic logic unit. *WSEAS Transactions on Biology and Biomedicine*, 1:436–440, 2004.

[19] Y. Tokumaru. Studies on procedures for floating point arithmetic operations with DNA molecules. Master's thesis, Kyushu Institute of Technology, 2007.

[20] H. Yoshida and A. Suyama. Solution to 3-SAT by breadth first search. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 54:pp.9–22, 2000.