Joint NCC & IIS Bull., Comp. Science, 4 (1996), 27–42

Two new cellular pipelined algorithms for computing a sum of products

V. Markova

Two new cellular pipelined algorithms (2D and 3D) for computing a sum of products with a very short multiplication time (six and three steps, respectively) are proposed. The initial data and results are binary signed-digit integers. The design and investigation tool is an experemental computer simulating system (Animating Language Tools), based on an original model of distributed computation (Parallel Substitution Algorithm).

1. Introduction

Designing fast algorithms for computing a sum of products today has arisen a great theoretical and practical interest, as the fast massive multiply-add computation is the main problem in digit signal processing and matrix operations. In this paper we develop two new algorithms (2D and 3D) for computing a sum of products that has a very short multiplication time (six and three steps, respectively) due to the following features:

- harnessing the binary signed-digit number system [1, 2];
- pipelining at both the initial data and the computation process levels;
- loading of the initial data, transformation of the intermediate results and thier moving in parallel.

In both algorithms the multipliers are loaded digit serially, the multiplicands – digit parallelly, the results are produced digit parallelly. A doubling of the multiplication time in the second algorithm is achived due to pipelining of the computation process in the third dimension. It is based on the stratification of 2D pairwise summation of the intermediate results (partial products and their sums) from the first algorithm, that is, on the execution of the pairwise summation in four layers of 3D array. The idea of the stratification consists in the replacement of the neighborhood in the rows in 2D array by the neighborhood in the layes in 3D one. As a result, a one-layer summation is transformed into a four-layer one.

The presented algorithms calculate a sum of m products of n-digit binary signed-digit integers in time equal to $n+6m+1$ and $n/2+2\log_2 n+3m+1$, respectively.

The design and investigation tool is the computer simulating system Animating Language Tools (ALT) [4]. It realizes all properties of an original model of cellular computation named Parallel Substitution Algorithm (PSA) [3]. PSA is specified by a set of parallel substitutions which operates over a cellular array in parallel (everywhere and at the same time). ALT system consists of the graphical means and the language for representing of parallel algorithms. It is the graphical representation of both cellular arrays and substitutions which allows us to construct easily, to modify a parallel algorithm, and to watch what is happening at any cell of all arrays during computational process. Moreover, ALT system checks out a parallel algorithm and determines its time complexity.

This paper is organized as follows. The first section describes the main operations in the binary signed-digit number system. The basic features of PSA and ALT are given in the second section. The third section presents the 2D cellular pipelined algorithm for computing sum of products and its time complexity. 3D cellular pipelined algorithm for computing sum products and its time complexity are described in the fourth section.

2. Binary signed-digit number system

2.1. Binary signed-digit representation

Binary signed-digit (BSD) number system falls in a class of redundant. It is defined as a positional number system with the radix $r = 2$ and the base $B = \{\overline{1}, 0, 1\}$ ($\overline{1} = -1$). Due to the redundancy any number can be represented in the BSD number system more than by one way. For example, the interger "7" has the following *binary signed-digit* representations (numbers): $0111, 100\overline{1}, 10\overline{1}1, 1\overline{1}11$.

The important characteristics of the BSD number system are as follows:

- the sign of the algebraic value is the sign of the most significant nonzero digits;
- the algebraic value of binary signed-digit integer is zero iff all its digits have zero value;
- the negation of binary signed-digit integer is directly defined by changing the signs of all nonzero digits in the integer;
- the carry propagates to one digit position.

The conversion of n-bit binary integer $\mathbf{X} = \mathbf{x}_{n-1}\mathbf{x}_{n-2}\ldots\mathbf{x}_0$ into n-digit BSD integer $X = x_{n-1}x_{n-2}\ldots x_0$ is very simple. If integer $\mathbf{X}$ is an unsigned binary integer, then $x_j = \mathbf{x}_j$ for all $j = 0, 1, \ldots, n-1$. If integer $\mathbf{X}$ is two's complement integer, then $x_{n-1} = \overline{\mathbf{x}}_{n-1}$ for $\mathbf{x}_{n-1} = 1$, otherwise $x_{n-1} = 0$, and $x_j = \mathbf{x}_j$ for all $j = 0, 1, \ldots, n-2$.

The conversion of n-digit BSD integer X into n-bit binary integer $\mathbf{X}$ is carried out as follows. Integer X is decomposed into two components X^+ and X^-, where X^+, X^- – unsigned integers formed of the positive and the negative digits of X, respectively. Then X^- is substructed from X^+, the obtained $(n+1)$-bit result is two's complement integer.

2.2. The main operations in binary signed-digit number system

Let $X = x_{n-1}x_{n-2}\ldots x_0$ and $Y = y_{n-1}y_{n-2}\ldots y_0$ be two binary signed-digit integers.

Binary signed-digit (carry-free) addition is performed in two steps. In the first step, we determine *the intermediate carry* digit c_{j+1} and *the intermediate sum* digit s'_j at each digit position j by examining digits x_j, y_j, x_{j-1} and y_{j-1} according to Table 1. In the second step, we calculate *the final sum* digit s_j at each digit position j by adding the intermediate sum digit s'_j and the intermediate carry digit c_{j-1}, without generating the carry. Thus, in the binary signed-digit addition the carry propagation is limited to a single digit position irrespective of the wordlength. Each sum digit s_j depends only on six digits: x_j, y_j, x_{j-1}, y_{j-1}, x_{j-2} and y_{j-2}.

Table 1

x_j	y_j	x_{j-1}, y_{j-1}	c_{j+1}	s'_j
0	0	Both are of any sign	0	0
0	1	Both are nonnegative	1	$\bar{1}$
1	0	Otherwise	0	1
0	$\bar{1}$	Both are nonnegative	0	$\bar{1}$
$\bar{1}$	0	Otherwise	$\bar{1}$	$\bar{1}$
1	$\bar{1}$	Both are of any sign	0	0
$\bar{1}$	1	—"—	0	0
1	1	—"—	1	0
$\bar{1}$	$\bar{1}$	—"—	$\bar{1}$	0

The above addition is called *the carry-free* addition. The carry-free addition property of the BSD number system is based on the recording of all digit pairs of the summed integers according to Table 1 in such a way that in the second step the final summation never generates the carry.

Example 1. Let $X = \bar{1}01011110$, $Y = \bar{1}1000000\bar{1}$ ($\mathbf{X}_{10} = -162$, $\mathbf{Y}_{10} = -127$). The computation of the BSD sum is shown Figure 1. It is easy to check that the obtained sum agrees with the sum computed directly. Indeed,

$$
\begin{array}{rcccccccccc}
 & \bar{1} & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & X \\
+ & \bar{1} & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \bar{1} & Y \\
\hline
 & 0 & \bar{1} & \bar{1} & 0 & \bar{1} & \bar{1} & \bar{1} & 1 & \bar{1} & S' \\
+ & \bar{1} & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & C \\
\hline
 & \bar{1} & 1 & 0 & \bar{1} & 1 & 0 & 0 & \bar{1} & 1 & \bar{1} \quad S
\end{array}
$$

Figure 1

the inverse conversion of the signed-digit integer S is $\mathbf{S}_{10} = \bar{1}(2)^9 + 1(2)^8 + \bar{1}(2)^6 + 1(2)^5 + \bar{1}(2)^2 + 1(2)^1 + \bar{1}(2)^0 = -289$.

Binary signed-digit multiplication is calculated in a classic manner. Firstly, n partial products are generated, and secondly, they are summed up.

All digits of *the partial product* $P_i = p_{i,n+1}p_{i,n}\ldots p_{i,0}$, $i = 0, 1, \ldots, n-1$, are calculated in parallel and take the following values:

$$
p_{ij} = x_j \cdot y_i = \begin{cases} 1, & \text{if } x_i = y_j = 1 \text{ or } x_i = y_j = \bar{1}, \\ \bar{1}, & \text{if } (x_i = 1 \text{ and } y_j = \bar{1}) \text{ or } (x_i = \bar{1} \text{ and } y_j = 1), \\ 0, & \text{otherwise.} \end{cases}
$$

Then the partial products are added up by means of the binary tree using the BSD addition. Computations at each *k-th* level of the tree, $k = 0, 1, \ldots, \log_2 n - 1$, are done in parallel. The sum obtained at the level $k = \log_2 n - 1$ is the BSD product P.

Example 2. Figure 2 shows the multiplication of two BSD integers $X = \bar{1}001$ and $Y = 0101$.

$$
\begin{array}{rcccc}
 & & \bar{1} & 0 & 0 & 1 & X \\
 & \times & 0 & 1 & 0 & 1 & Y \\
\hline
 & & \bar{1} & 0 & 0 & 1 & \\
 & + & \bar{1} & 0 & 0 & 1 & \\
\hline
 & \bar{1} & 0 & 1 & 1 & 0 & \bar{1} \\
 & + & & & \bar{1} & 0 & 1 \\
\hline
 & \bar{1} & 1 & 0 & 1 & 0 & 0 & \bar{1} & 1 & P \\
\end{array}
$$

Figure 2

3. Computer simulating system ALT

Animating Language Tools (ALT) [4] is the computer simulating system intended for design of cellular algorithms and determination of thier time

complexity (number of steps needed for obtaining the result). It realizes all properties of original model of cellular computation named Parallel Substitution Algorithm (PSA) [3].

3.1. Basic features of PSA

Bellow we briefly remind the basic features of PSA.

- PSA processes data represented by *a cellular array* W. Each cell is a pair (a, m), where a is *the state* from given finite alphabet A, m is *the name* from a set of names M (as a rule, names are the coordinates in p-dimensional discrete space, $p = 1, 2, 3$).
- Processing of a cellular array is performed by a set of *substitutions*

$$W_1 * W_2 \to W_3, \tag{1}$$

where W_1, W_2, W_3 are cellular arrays, W_1 – the base, W_2 – the context, $W_1 * W_2$ – *the left-hand* side and W_3 – *the right-hand* side. The substitution (1) is *applicable* to the array W if $W_1 \cup W_2 \subseteq W$. Each side of a substitution is *a pattern*, it points out space relations between *the neighboring* cells, i.e., cells, which participate in data processing. *The execution* of an applicable substitution is the replacement of the array W_1 by the array W_3. To map the applicability of a substitution (1) to each cell (a_i, p_i), $i = 1, 2, \ldots, t$, in W, the certain names in the arrays W_1, W_2, W_3 are specified by *naming functions* $\varphi(m) : M \xrightarrow{b} M$, the arrays are specified by *configurations* $S = \{a_i, \varphi_i(m)\}$, $\varphi_i(m) \neq \varphi_j(m)$, $i \neq j$. For certain $m_g \in M$ a configuration defines a subarray of *the neighboring* cells. In terms of configurations a parallel substitution is as follows:

$$S_1 * S_2 \to S_3. \tag{2}$$

Substitution (2) is applicable to W, if there exists at least one $m_g \in M$, such that $S_1(m_g) \cup S_2(m_g) \subseteq W$. The execution of an applicable substitution is the replacement of the subarray $S_3(m_g)$ by the subarray $S_3(m_g)$, i.e., the base cells change their states. This replacement is performed simultaneously for all sets of the neighboring cells which meet the applicable conditions. All substitutions are applied in parallel (at once) at each cell of the array W.

- The computation is an iterative procedure. At each step all applicable substitutions are executed resulting in a new cellular array. This procedure is repeated until we arrive at such cellular array to which no single substitution is applicable.

3.2. Computer simulating system ALT

Bellow we present only a kernel of ALT system: the graphical representation of cellular arrays and substitutions, and the main operators of the ALT-language.

The graphical representation of cellular arrays and substitutions. A cellular array is pictured a bundle of sheets, only the 0-th layer being entirely seen. The left-hand and right-hand sides of substitutions are displayed as 1D, 2D or 3D right-angled patterns (a 3D pattern is considered as a multilayered structure). The coordinate origin in cellular arrays and patterns is always chosen in the left-top cell which is closest to the observer. The neighboring cells in the patterns are labelled by variable symbols, whose domain is in the initial alphabet A, or the fixed color corresponding to the given symbol of the alphabet A. In other cells "don't care" symbols are written. In the arrays cells are labelled by fixed color or "don't care" symbols. Each cellular array and pattern are assigned by a unique names. They are displayed in a special window on the monitor screen. The graphical representation not only of cellular arrays but also of the both sides of substitutions is the peculiarity of this system. It is the graphical representation which allows us to construct easily and to modify the patterns, and, moreover, to watch what is happening at any cell of the array during computational process.

The main operators of the ALT-language. Data processing in arrays is specified by the program scheme which is also displayed in a special window on the monitor screen. The following operators are used in the program scheme.

- The operator **ex** specifies the synchronous and the iterative execution of the PSA.
- The operator **in** followed by the name of the array specifies the space, where the substitutions are to be executed. It is possible to use a composition of severel arrays. In this case their names are listed separated by the symbol $*$.
- The ordinary substitution is given by two operators **at** and **do**. The word **at** preceeds the pattern name from the left-hand side of the substitution (or the list of pattern names separated by the symbol $*$ and located as well as the names of the arrays in the operator **in**). The word **do** preceedes the pattern name from the right-hand side of the substitution (or the list of pattern names separated by the symbol $*$ and located in the same order that the names of the arrays in the operator **in**.
- The functional substitution is given by two operators **ab** (instead of the operator **at**) and **do**. The operator **do** is followed by the name

of the function. It calculates new values of variable symbols. Each function is an expression of C-language operator. It is shown in the special window.

A representation of PSA in ALT is called *ALT-model.*

Below the design of the cellular algorithm is demonstrated by the example of the binary signed-digit summation of many integers.

Example 3. The summation is performed in arrays named p and c with the sets of names $\{\langle i,j\rangle : i = 0,1,2,3,\ j = 0,\ldots,8\}$, and $\{\langle i,j\rangle : i = 0.1,2,3,4,\ j = 0,1\}$, respectively. Their initial states are shown in Figure 3a (the symbol $\times$ denotes "don't care" symbol). The array p is the processing one. Two processes (the summation and the data shift in the direction of the larger value of i) are performed in it. Each row of the array p in the initial state stores one out of four integers, the least significant digit is placed in the j-th column. The cell states in the array p takes the values from the base. The sum is generated in the 3-rd row of p. The array c is the controlling one. The array c_0 (the 0-th column without the top cell) distinguishes the even and the odd rows in *pa.* The array c_1 (the 2-nd column without the top cell) generates the signals for performing one out of two steps of the addition and a data shift. The cell states in the array c are from the set $\{1.2.3,4\}$.

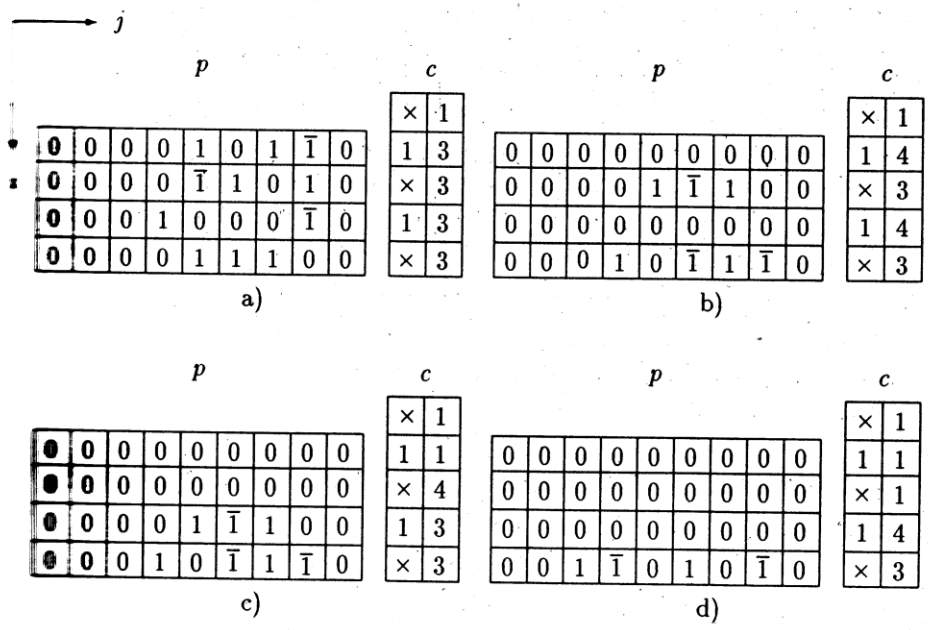

Figure 3

How does this algorithm work? At the fist two steps, the algorithm carries out the BSD summation of two pairs of neighboring integers of the processing array. (Here neighboring integers are located in two rows, the first having the even and the second having the odd numbers.) Each summation step is accompanied by the cell states changing in c_1. At the 3-rd step, the algorithm moves the top sum of the array p to the 2-nd row. Further the summation is performed in two latter rows. The cells $(1, \langle 3, 0\rangle)$, $(3, \langle 3, 1\rangle)$, and $(3, \langle 4, 1\rangle)$ in the array c form the applicability condition for performing the 1-st step of the summation (in bracets the first symbol stands for the state, the second symbol – the name of the cell). The results of the 2-nd, 3-rd and 5-th (the last) steps are shown in Figures 3b, 3c and 3d, respectively.

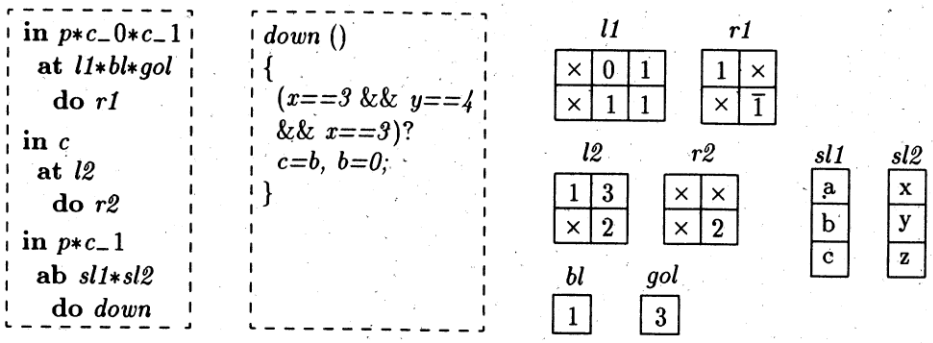

Figure 4

In Figure 4 the fragment of the ALT-model of the BSD summation is presented, using the patterns *l1, r1, l2, r2, bl, gol, st1, st2.* The 1-st pair (**at-do**) is one out of the forty one substitutions realizing the 1-st step of the summation. The patterns *l1, bl, gol*, whose names are pointed out in the operator **at**, are recognized in the composition of the arrays $p*c\_0*c\_1$. The entry of those patterns is thought to be recognized, if the pattern *l1* is included in the array p, the pattern *bl* – in the array c_0, the pattern *gol* – in the array c_1, and, moreover, the coordinates of the left-top cells of those patterns coincide. The 2-nd pair (**at-do**) is the substitution generating the signal in the array c for the 2-nd step of the summation. The pair (**ab-do**) is the data shift substitution in the processing array.

4. 2D cellular pipelined algorithm for computing a sum of products

In this section, we present 2D cellular algorithm for computing a sum of products and evaluate its time complexity (*the multiplication time* T_m and *the execution time* T_e). The former is defined as a time between two succes-

sive calculations of products, the latter – as the total time to genarate the result.

4.1. 2D Cellular pipelined algorithm for computing a sum of products

2D cellular pipelined algorithm for computing a sum of products is carried out in the arrays *y, x, p1, p2, c1, c2, c3.* The initial state of those arrays for summing three products of 8-digit integers are shown in Figure 5. The arrays y and x store the initial data (the multipliers and the multiplicands). The least significant digit of the multiplier is placed in the 0-th row of y, the least significant digit of the multiplicand – in the 16-th column of x. The first pair of the initial data (Y_0, X_0) is located in the arrays y_y (the 3-rd column of y) and p_x (the 0-th rows of *p1*). The arrays *p1* and *p2* are intended for processing: the former calculates products, the latter – a sum of products. The cell states in *y, x, p1, p2* take the values from the base. The array *c1* controls the generating of partial products and multiplicand shifting in *p1* as well as the loading of a new multiplier digit serially in y_y and a new multiplicand digit concurrently in p_x. Two columns from the array *c2* play the same role that the array *c* from Example 3. The cell named $\langle 8, 2\rangle$ from the array *c2* generates the signal indicating that the BSD

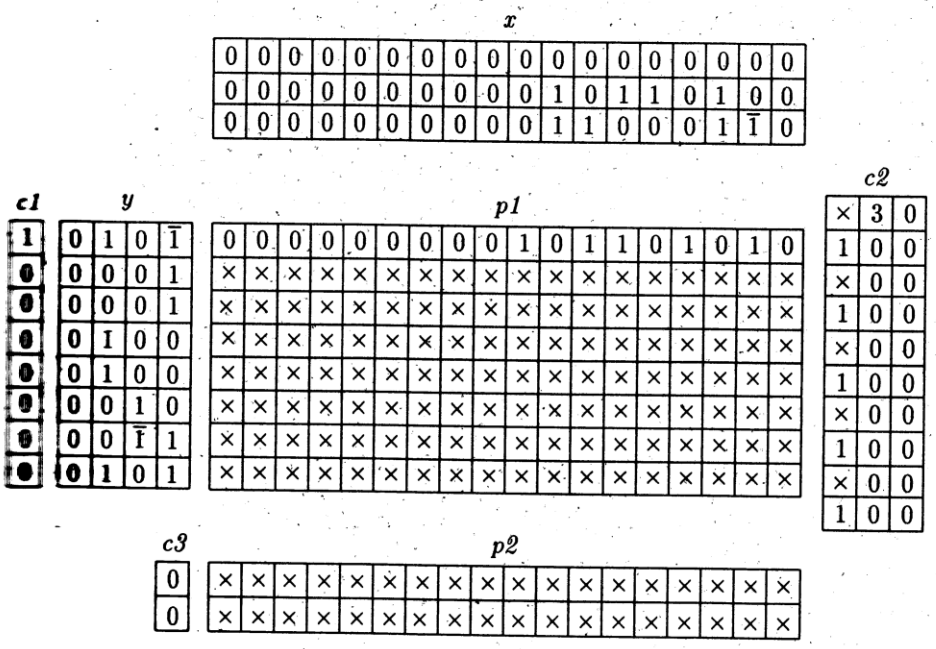

Figure 5

product is obtained, and then it is transfered to in the array *p2* under the control of the array *c3*. Moreover, this array controls the BSD summation in the array *p2*. The cell states in *c1* are from the set $\{0,1\}$, in *c2* and *c3* – from the set $\{0,1,2,3,4\}$.

To achive a short mulptiplication time, the computation process in the algorithm is organized as follows.

- Loading of a new multiplier digit in $y-y$ is done at each step. Loading of a new multiplicand in p_x is performed at six step intervals, begining from the second multiplicand.
- Loading of the initial data, transformation of the intermediate results (generation of partial products and the BSD summation) and thier moving are carried out in parallel. Moreover, the 2-nd step of summation in h-th, $h = 0,1,\ldots,n/2-1$, pair of the neighboring integers is merged with shifting the obtained sum one position down, if in $(h+1)$-th pair of the neighboring integers the 2-nd step of summation is aslo under execution.

4.2. The time complexity of 2D cellular pipelined algorithm for computing a sum of products

In Figure 6 the dependence graph of multiplication algorithm for three pairs of 8-digit integers is shown. Here the symbols c, s, $\bullet$, $\circ$ and $*$ stand for the carry, the sum and three BSD integers (partial products and their sums), respectively.

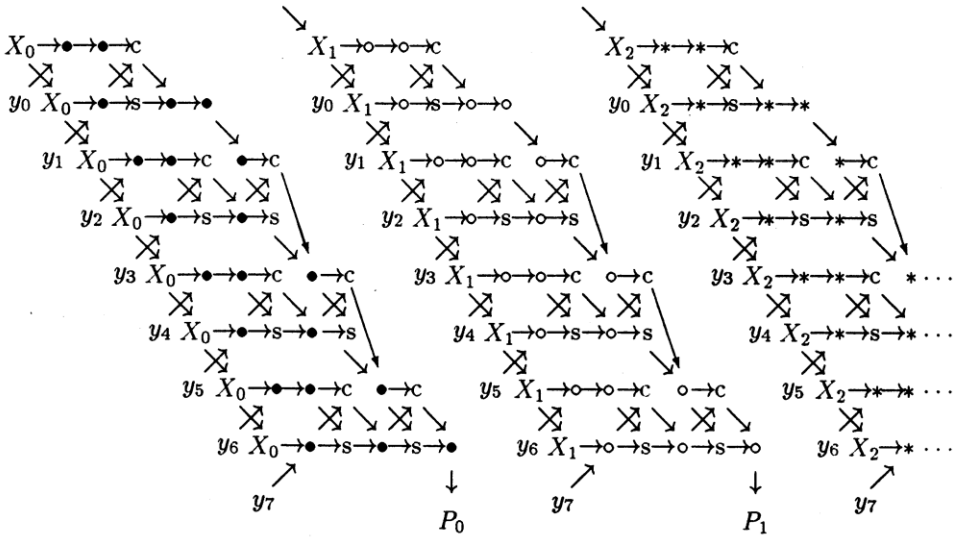

Figure 6

Let us consider the calculation of the 0-th product P_0. As distinct from the summation by means of the binary tree, i.e., the serially-parallel scheme, here the summation is performed for all those pairs of the neighboring integers, which meet the applicable conditions. The execution in h-th pair of neighboring integers precedes (with overlaping) the execution in $(h+1)$-th pair. Each pair is involved in the product calculation during a constant time independent of the word lenght of an integer. Begining from the 2-nd pair, this time is 6 steps for the 1-st integer and 5 steps for the 2-nd one. Hence, loading of the multiplicand X_1 is performed at the 6-th step, the time, needed for obtaining the product P_0, is $T = n+4$ for n-digit integers.

So, the presented 2D algorithm has the following time complexity:

(a) the multiplication time is a constant ($T_m = 6$), independently of the lenght of an integer;

(b) the execution time is $T_e = (n+4)+6(m-1)+3 = n+6m+1$, where m is the number of products.

This algorithm and the algorithm realized in a classic manner, i.e., in which shifting the obtained sum is performed after the 2-nd step of summation, has been simulated for same values of n (from 16 - to 64) with the help of ALT. As distinct from the former, in the latter the multiplication time is dependent of the length of an integer:

n	16	24	32	48	64
T_m	12	13	15	16	18

5. 3D cellular pipelined algorithm for computing a sum of products

Further way to improve the time complexity of the above algorithm is the pipelining of the summation process in the third dimension. It is achieved due to the stratification of 2D pairwise summation of the intermediate results, namely, the execution of the pairwise summation in four layers of 3D array. In this section, we describe the idea of the stratification, at first, and then the 3D algorithm and its time complexity.

5.1. Idea of the stratification of 2D pairwise summation

The stratification of 2D pairwise summation is based on the replacement the neighborhood in the rows in the 2D pattern by the neighborhood in the layers in the 3D one (Figure 7). As a result, a 1-layer pattern is transformated into a 4-layer one. A pair of integer digits to be summed up and the results of the 1-st summation step are placed in the 0-th and the 1-st layers of the

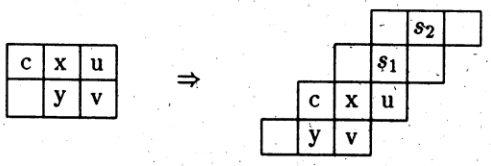

Figure 7

4-layer pattern. The final sum digits (s_1, s_2) are written in the 2-nd or 3-rd layers of the pattern. The pairwise summation is performed, using the 4-layer pattern, further reffered as *the four-layer summation.*

5.2. 3D cellular pipelined algorithm for computing a sum of products

The algorithm is carried out in 3D arrays *x, y, p1, p2, c1, c2* and *c3.* The initial state of those arrays for computing a sum of three products of 8-digit BSD integers are shown in Figure 8. The array y has $n+1$ rows, 4 columns and 3 layers. The 0-th layer of y stores the set of multiplier. The array x

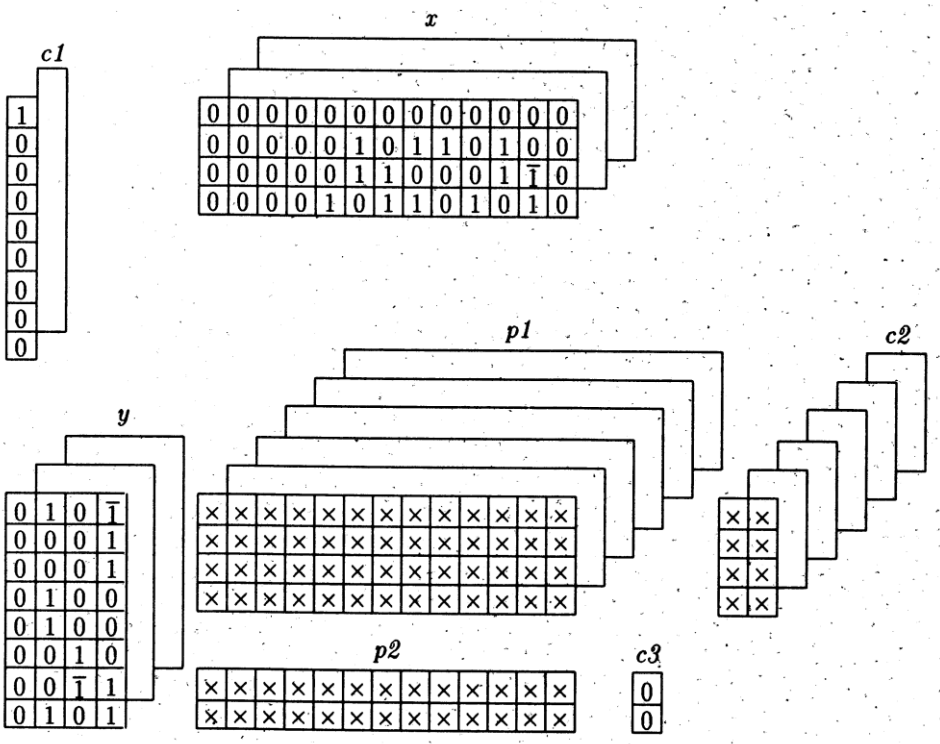

Figure 8

has 4 rows, $2n + 5$ columns and 3 layers. The 0-th and the 1-st layers of x store two sets of multiplicands $X^{\leftarrow}$ and X, the first shifted one position left relative to the second. In both arrays the 2-nd layer is the controlling one. The arrays *p1* and *p2* are intended for processing. The former has $n/2$ rows, $2n+5$ columns, and $2\log_2 n$ layers. Products are calculated in it. The latter has 2 rows and $2n + 5$ columns. A sum of the obtained products is defined in it. The array *c1* controls the generating of partial products and multiplicand shifting in *p1* as well as the loading of the initial data. It has $n + 1$ rows, 1 column and 2 layers. The control for computation process in the array *p1* is carried out the array *c2*. It has $n/2$ rows, 2 columns and $2\log_2 n$ layers. The array *c3* controls the BSD summation process in the array *p2*. The presented algorithm consists of two threads.

The first thread transforms the multiplier set into two subset, located in the 0-th and the 1-st layers of the array y. This transformation for one multiplier is illustrated in Figure 9. Here the symbols ♠ and ♡ stand for even and odd digits. The transformation consists in the following. At the 1-st step, in the array y all even multiplier digits are dropped from the 0-th layer to the identical rows of the 1-st layer, all odd digits are shifted one digit to the top of the 0-th layer. At the 2-nd step, the 0-th pair of multiplier digits is fed to the 0-th rows of the 0-th and the 1-st layers of the array *p1*, the other pairs are shifted one position to the top. Beginning with the 3-rd step, the moving digit pairs in y is continue. Each j-th pair reaches to j-th row of the 0-th and the 1-st layers of the array y and is fed to the identical rows and layers of the array *p1*. The transformation of the 2D multiplier array is finished at the $(n/2 + 1)$-th step.

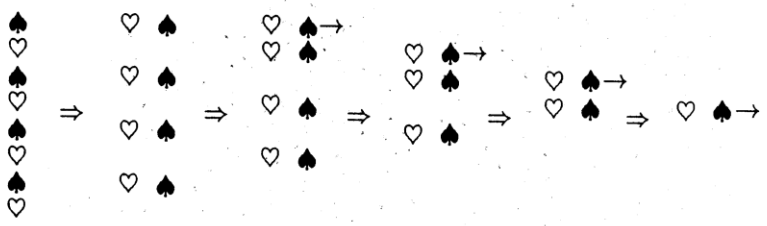

Figure 9

The second thread carries out the BSD multiplication itself in the array *p1*. The main procedure in this thread is the four-layer summation. For simplicity Figure 10 gives the graphical representation of this thread for multiplying three pair of 8-digit integers. Here, using three symbols •, ○ and ⋆, the propagation of three computation fronts is shown. Each above symbol stands for the intermediate results obtained by calculating one out of three products P_0, P_1, P_2.

So, the array *p1* in this example has 1 column, 4 rows and 6 layers. At the 1-st step of the second thread, two first multiplicands belonging to

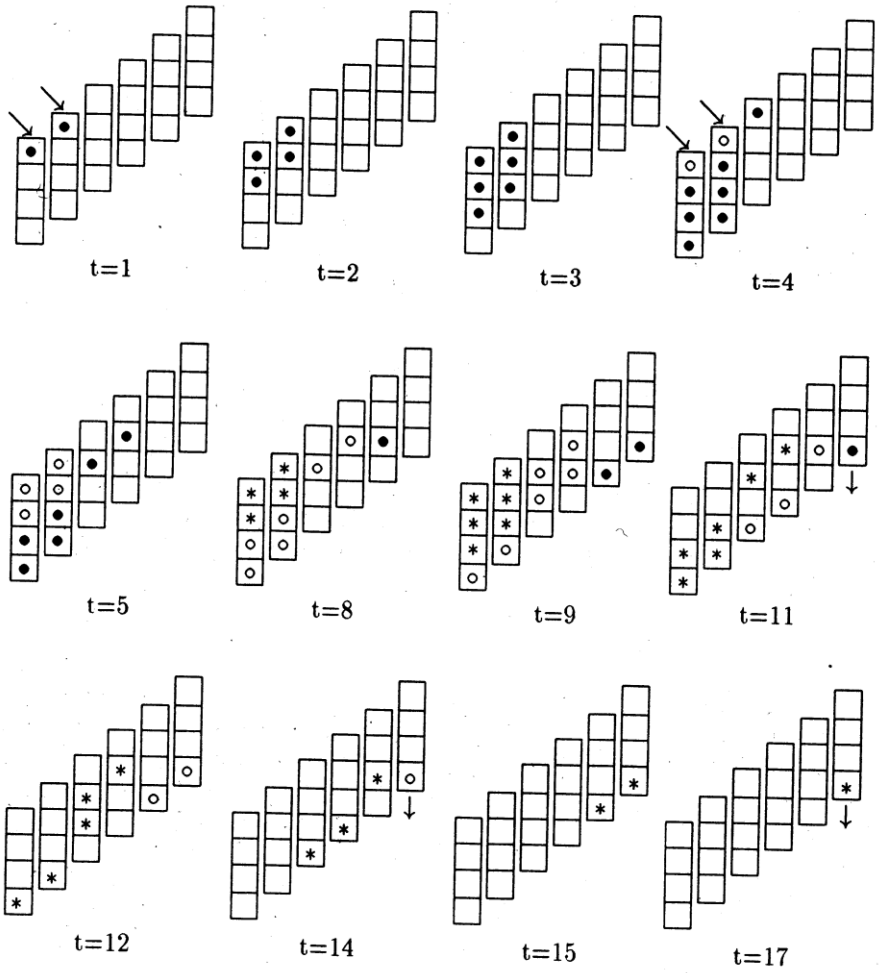

Figure 10

$X^{\leftarrow}$ and X from the array x are loaded into the 0-th rows of the 0-th and the 1-st layers of the array *p1* concurrently. At the 2-nd step, two partial products are calculated in the 0-th rows of the 0-th and the 1-st layers, and multiplicands are shifted one position to the bottom in the 0-th and the 1-st layers of *p1*. The result of the 4-th step of this thread are the first BSD sum in the 0-th row of the 2-nd layer of the array *p1*, the first intermediate carry and sum in the 1-st rows of the 0-th and the 1-st layers, two partial products in the 2-nd rows of the same layers and two first multiplicands in the 3-rd rows. At the same time, the next two multiplicands are loaded into the 0-th rows of the 0-th and 1-st layers of the array. At the 5-th step, leaving out the calculations in the first two layers, the second BSD sum is calculated and located in the 1-st row of the 3-rd layer. At the same time,

the first BSD sum is shifted into the 1-st row of the 2-nd layer, as a result of which, the pair of integers to be summed up is formed in the 2-nd and the 3-rd layers. The multiplication process is repeated, involving new layers of *p1* and new multiplicands, until the last BSD product is obtained in the 3-th row of the 5-th layer.

As one can see, the above algorithm calculates the first product in time $T = (n/2 + 1) + 2\log_2 n$ for n-digit numbers. In our case, the first product is computed at the 11-th step.

This algorithm has been simulated for same values of n (from 16 to 64) with the help of ALT.

5.3. The time complexity of 3D cellular pipelined algorithm for computing a sum of products

The introduced time complexity estimates of the presented 3D algorithm are as follows.

- The multiplication time is a constant ($T_m = 3$).
- The execution time is $T_e = (n/2 + 1 + 2\log_2 n) + 3(m - 1) + 3 = n/2 + 2\log_2 n + 3m + 1$, where m is the number of products.

6. Conclusion

In this paper we persent two new cellular pipelined algorithms (2D and 3D) for computing a sum of products. They have the following characteristics.

- The initial data and the results are binary signed-digit integers. The multipliers are loaded digit serially, the multiplicands – digit parallelly, the results are produced digit parallelly.
- The time complexity estimates of the presented algorithms are listed in Table 2.

Table 2

Algorithm	T_m	T_e
2D	6	$n + 6m + 1$
3D	3	$n/2 + 2\log_2 n + 3m + 1$

As would be expected, the 3D algorithm calculetes a sum of products in better time that presented here 2D one. The speed-up is achieved due to pipelining of the computation process in the third dimension. However, it is difficult to compare two algorithms with respect to the implementation (area) complexity without doing detailed designs.

References

[1] A. Avizienis, *Signed-digit number representations for fast parallel arithmetic*, IRE Trans. Electron. Comput., **EC-10**, 1961, 389–400.

[2] K. Hwang, A. Louri, *Optical multiplication and division using modified-signed symbolic substitution*, Opt. Eng., **28**, No. 4, 1989, 364–372.

[3] S.M. Achasova, O.L. Bandman, V.P. Markova, S.V. Piskunov, *Parallel Substitution Algorithm*, World Scientific, Singapore, 1994, 220 p.

[4] Yu.M. Pogudin, *ALT – a graphical system for parallel microprogramming*, Parallel Algorithms and Structures, Computer Center, Novosibirsk, 1991, 77–88 (in Russian).