Array Processor Featuring an Effective FIFO-based Data Stream Management

Toshiaki Miyazaki, Yuusuke Nomoto, Yuka Sato, and Stanislav G. Sedukhin
Graduate School of Computer Science and Engineering, The University of Aizu
Aizu-Wakamatsu, Fukushima 965-8580, Japan
E-mail: miyazaki@u-aizu.ac.jp

Abstract

In array processors, data I/O management is the key to realizing high-speed matrix operations that are often required in signal and image processing. In this paper, we propose an array processor utilizing an effective data I/O mechanism featuring external FIFOs. The FIFOs are used to buffer initial matrix data and partially processed results. Therefore, if all required data are stored in the FIFOs, matrix operations, including the algorithm to solve the Algebraic Path Problem (APP), can be performed without any data I/Os. In addition, we can eliminate register files from the processing elements (PEs) if we construct the PE array by controlling the external FIFOs systematically and transferring the data from (to) the FIFOs to (from) the PE array. This enables us to simplify each PE structure and realize a large array processor with limited hardware resources. Here, the FIFOs themselves can be easily realized using conventional discrete FIFO or memory chips.

1. Introduction

It is well known that data-level parallelism is inherent in many scientific, engineering, signal, and media processing algorithms as these algorithms typically repeat a small set of operations over a sequence of points in grids, input pixels, video frames, or sound samples. This data-level parallelism is typically expressed in a matrix form. Thus, matrix manipulations are very important, and many types of algorithms and implementations dedicated to these manipulations have been proposed previously [1][2]. A systolic array [3] is one of the well-known array processors. It consists of many basic processing elements (PEs) and has several variations such as a linear array and a mesh array that differ according to the manner in which the PEs are connected. Because of its simple structure, the systolic array is relatively easy to implement. However, the input data must be promptly fed to the systolic array, and sometimes the output data from the array needs to be fed repeatedly to perform the expected calculations. Thus, the data manipulation mechanism is more complicated as compared to the implementation of the systolic array. Hence, the data manipulation mechanism performed outside the array could be a performance bottleneck for the systolic array-based processors.

To resolve this problem, we first propose a basic algorithm using a 2D array structure. This algorithm does not require any complicated data management mechanism because it stores all input data inside the array and efficiently performs matrix-matrix operations by rotating the stored data among the PEs without any data I/O operations during processing. The 2D array processor has a 3-input fused operation unit (FOU) and a register file (reg-file) in each PE and can solve many problems related to matrix-matrix operations including the Algebraic Path Problem (APP). The only problem of the array processor is that if the size of the array is smaller than that of the input matrices, each PE must have a large reg-file to store all input matrices. Subsequently, we introduce an effective data stream management mechanism using FIFOs, named "*QueueDoRegister*" (external queues realizing pseudo register file). By introducing this mechanism to a basic 2D array processor, we can handle large matrix operations without modifying the original procedure to solve problems. The merits of this data management mechanism are as follows:

1. By storing the matrix data into FIFOs, each PE does not require a large reg-file even if the input matrix is large.
2. By separating the data storage part from the PE array as FIFOs, each PE is simplified and a large number of PEs can be realized on one chip. In addition, the FIFOs can easily be realized with commercially available discrete FIFO or memory chips. Consequently, a large array processor system can be implemented with some board-level integration techniques.

In this paper, we concentrate on how to apply the new data I/O mechanism to a basic 2D array processor dedicated to the APP. However, we believe that this I/O mechanism can also be applied to other array processors such as the conventional systolic array.

This paper is organized as follows. In Section 2, the APP, i.e., our target, and an algorithm to solve it with the basic 2D array structure are briefly introduced. Next, the new array processor architecture featuring the *QueueDoRegister* mechanism is explained in Section 3. Then, some evaluations and remarks are described in Section 4. Finally, the conclusions are described in Section 5.

2. Algebraic path problem

Before introducing our architecture, we would like to describe the APP, which is one of our targets, and the manner in which it can be solved using our array processor. The APP is a general framework that unifies several solution procedures for a number of well-known matrix and graph problems such as matrix inversion, transitive closure, all-pairs shortest paths, and minimum spanning tree. Hence, if a common effective mechanism is provided to solve the APP, we can apply it to many real-world problems.

Consider a weighted directed graph $G = (V, E, w)$, where $V = \{0,1,...,n-1\}$ is a set of n vertices, $E \subseteq V \times V$ is a set of edges, and $w: E \to S$ is an edge weighting function whose values are obtained from the set S. This function belongs to the path algebra $\langle S, \oplus, \otimes, (*), \bar{0}, \bar{1} \rangle$ together with two operations—"addition" $\oplus : S + S \to S$ and "multiplication" $\otimes : S \times S \to S$, and a unary operation called "closure" $(*) : S \to S$. The constants $\bar{0}$ and $\bar{1}$ belong to *S*. This path algebra is a closed semiring.

A path p is a sequence of vertices $(v_0, v_1, ..., v_{t-1}, v_t)$, where $0 \le t$ and $(v_{i-1}, v_i) \in E$. The weight of the path p is given as follows:

$$w(p) = w_1 \otimes w_2 \otimes ... \otimes w_t,$$

where w_i is the weight of the edge (v_{i-1}, v_i).

The APP is defined as the determination of the sum of weights of all possible paths between each pair of vertices (i, j). If $P(i, j)$ is the set of all possible paths from *i* to *j*, then the APP determines the values as follows:

$$d_{ij} = \oplus \{ w(p) : p \in P(i, j) \}.$$

The detailed discussion of the APP is beyond the scope of this paper and can be found in [4][5][11]. Here, if we map $\oplus$, $\otimes$, (*), $\bar{0}$, and $\bar{1}$ to operations and values, as shown in Table 1, we can solve many problems using the same algorithm. This implies that a hardware mechanism to solve the APP can be used in many applications. From the viewpoint of hardware implementation, this is a great merit.

The APP may be formulated in a matrix form. In this case, we associate the $n \times n$ matrix $A = [a_{ij}]$ with the weighted graph $G = (V, E, w)$, where $a_{ij} = w(i, j)$ if $(i, j) \in E$ and $a_{ij} = \bar{0}$ otherwise. This matrix representation allows us to formulate the APP as the computation of a sequence of matrices $A^{(k)} = [a_{ij}^{(k)}]$, $1 \le k \le n$. The value $a_{ij}^{(k)}$ is the weight of all the possible paths from vertex *i* to vertex *j* with intermediate vertices *v*, $1 \le v \le k$. Initially, $A^{(0)} = A$; subsequently, $A^* = A^{(n)}$, where $A^* = [a_{ij}^*]$ is an $n \times n$ matrix satisfying the condition $a_{ij}^* = d_{ij}$ if $(i, j) \in P(i, j)$ and $a_{ij}^* = \bar{0}$ otherwise. Here, if the given matrix size $n \times n$ is larger than the size of the provided array processor $b \times b$, i.e., *if* $n > b$, then block matrix operations can be applied to solve the APP. Some APP-solving algorithms using block matrix operations can be found in [6][7][8][12]. The block algorithms are based on an extensive use of matrix-matrix multiply-add operations in different semirings, and each of these operations can be efficiently implemented in *b* steps on a 2D $b \times b$ torus array processor by using some algorithms that are similar to Cannon's algorithm [9]. In this case, the time complexity for solving the APP problem will be $O(n^3/b^2)$.

For an idempotent semiring ($a = a \oplus a$), the fine-grained version of the APP with the $b \times b$ array size can be represented as a special case of $b \times b$ matrix-matrix multiplication and solved in parallel on a $b \times b$ array processor by using the 3D three-pass procedure shown in Fig. 1. This procedure is a 3D generalization of the well-known GKT algorithm [10]. Here, we use as the input data a $b \times b$ weighted adjacency matrix $A = [a_{ik}] = A \oplus I$, where *I* is an identity $b \times b$ matrix in a corresponding semiring, $A' = [a'_{kj}]$ is a copy of matrix *A*, and matrix $C = [c_{ij}]$ is a matrix whose elements are initially equal to $\bar{0}$. This procedure assumes a 3D toroidal index space shown in Fig. 2. This is an example of the $b = 4$ case, i.e., a $4 \times 4 \times 4$ index space. Here, $i, j, k \in [0, b-1]$. An initial distribution of elements of *A*, *A'*, and *C* among the 3D toroidal $b \times b \times b$ index space is predefined by the selected time-step function step$(i, j, k) = (k - j - i)$ **mod** $b = 0$. The data elements associated to index points meeting the time-step function move every step in the 3D index space. At the beginning, the elements of the activated points where step$(i, j, k) = 0$ are initialized as $a_{ik} = a_{in}$, $a'_{kj} = a'_{in}$, and $c_{ij} = c_{in} = \bar{0}$. At each step$(i, j, k) = 0, 1, …, b-1$, in each index point $(i, j, k)^T$, a scalar fused multiply add (**fma**) operation (line 5 in Fig. 1) and a rolling action in which data is rolled to the neighboring points (lines 7–10 in Fig. 1) are synchronously performed. We call a pair of an **fma** operation and a data rolling action a "compute-and-roll" step. After *b* compute-and-roll steps, the time-step function becomes step$(i, j, k) = 0$ again, and all updated elements of *A*, *A'*, and *C* will be located at the same index points as they were initially, i.e., the data will be ready for the second and third passes.

Basically, each compute-and-roll operation can be realized by a single-instruction-multiple-data (SIMD)-based control. However, to perform the procedure shown in Fig. 1, an extra control mechanism is required. Here, we introduce Boolean attributes *r* and *s* and assign them to the elements of matrices *A* and *A'*, respectively. The values of *r* and *s* are $r = s = 1$ for the diagonal elements and $r = s = 0$ otherwise. During processing, these Boolean attributes are used to control different types of reassignments in each active index point $(i, j, k)^T$ in the 3D index space, as shown in Fig. 1. In other words, the output data of each index point are automatically changed according to the combination of the *r* and *s* values. Thus, these Boolean attributes eliminate an expensive checking of index conditions at each computing step.

As shown in Fig. 1, the **fma** operation is apparently the key to accelerate the APP-solving procedure. In all, three passes are needed to finalize the APP algorithm. Thus, $3b$ compute-and-roll steps are needed to solve the APP. At each step, b^2 compute-and-roll operations are performed and therefore a total of $3b^3$ fused operations are required to complete the APP algorithm. All the admissible 2D toroidal array processors can also be easily found by using appropriate 3D→2D projections. Figure 3 shows the 2D projection toward the k-axis of the 3D index space shown in Fig. 2. In this projection, the $(i, j)^{th}$ element of matrix *C*, i.e., the result, is stored and updated in the corresponding $(i, j)^{th}$ PE. Thus, our target is to realize a 2D array processor that has the structure shown in Fig. 3. Apparently, the processor can realize ordinary matrix manipulations such as matrix-matrix multiplication.

3. Architecture

As explained in the previous section, the APP can be solved effectively by using the 2D array processor shown in Fig. 3. However, if the array size is not sufficient for the given APP, we have to split the input matrix representing the APP into some partial matrices called "blocks" and solve the APP by applying an algorithm, called block algorithm, derived from the algorithm shown in Fig. 1 in order to handle the block matrices. Although the calculation steps would increase, we can solve a large APP with a small PE array by using the block algorithm. The only problem is that each PE must have a large reg-file to store the input data and partial results because each PE is repeatedly used to obtain a partial result from each block matrix, and these results must be retained in the PE to compute the final results of the block algorithm. To resolve this problem, we introduce a new data handling mechanism called *QueueDoRegister* to the basic array processor. This mechanism primarily buffers the data into the FIFOs and rotates them in the torus structure by connecting the PEs and FIFOs to each other during processing. Here, only some of the rotated data needs to be stored in the reg-files in PEs in the original block algorithm to solve the APP.

Figure 4 shows an overview of a new array processor featuring *QueueDoRegister.* It mainly consists of a PE array, external FIFOs, and a controller. The PE array has 2D torus connections, i.e., each PE has vertical and horizontal connections with the neighboring PEs, and the upper-side and left-side PEs are connected to the opposite-side PEs via the FIFOs. Here, FIFO_B and FIFO_C are used to store input matrices *B* and *C*, while FIFO_A is used to store input matrix *A*. The data buffered in FIFO_A and FIFO_B are fed to the PE array one by one in a manner similar to the systolic array. The outputs from the left-side and upper-side PEs are restored to the connected FIFOs and fed again to the PE array at appropriate times. FIFO_C is used to store matrix C when $C = \bar{C} \oplus A \otimes B$ is calculated. In the APP solving procedure shown in Fig. 1, *B* is *A'*, i.e., a copy of matrix *A*. FIFO_A actually consists of two parallel FIFOs—FIFO_A0 and FIFO_A1. Their usage is described in detail in Section 3.2.

The controller shown in Fig. 5 manages the PE array and the FIFOs according to the given instructions. Our array processor is basically controlled with an SIMD stile; however, the output data in each PE are selected by the Boolean attributes *r* and *s* attached to the elements of matrices *A* and *B* (= *A',* in the case of APP), respectively (see lines 7–10 in Fig. 1). In addition to an instruction register and a decoder, there is a counter that is used to decide how many times the instruction fetched in the instruction register should be issued. For example, if the fetched instruction should be issued five times continuously, the counter is set to 5. In matrix operations, the same operations often need to be repeated for all elements in the matrix. Thus, this mechanism is very useful in such cases.

An overview of each PE is shown in Fig. 6. It is made of a 3-input 1-output FOU, three registers (reg_a, reg_b, and reg_c), and a cross-bar switch. The registers reg_a and reg_b hold an element of the matrices *A* and *B,* respectively. For matrix *C*, reg_c is provided. Here, reg_c actually consists of two registers reg_c0 and reg_c1. The use of reg_c0 and reg_c1 is discussed later. As shown in Fig. 6, there is no large reg-file in the PE. After a 3-input operation in the FOU is performed, the result is stored in reg_c or transferred to the neighboring PEs. The cross-bar switch selects which values among those in reg_a, reg_b, and the FOU output to the upper and left PEs, according to the Boolean attributes *r* and *s* attached to each element of matrices *A* and *B*, they are also loaded to in reg_a and reg_b with the element values.

The FOU performs some type of scalar fused operations, as shown in Table 1. The most unique feature of this FOU is that it supports min/max (data comparison) and Boolean fused operations as well as the ordinary **fma** operation, which is often available with some commercial DSPs and MPUs. One of the fused operations is selected by the function select signal from the control unit in a manner that is similar to the selection process of a normal ALU control. To realize the FOU, we provided data comparators and Boolean operators in addition to a multiplier and an adder/subtractor, and tightly combined them together. Thus, any fused operation featured in the FOU can be performed with one system clock. It is very important to perform the compute-and-roll operation within a single clock cycle, regardless of the variations of the fused operations. In fact, it cannot be done with the existing array processors or DSPs/MPUs because the required data comparison in min/max operations often requests several instructions and clocks, i.e., at least a subtraction and a condition branch, if the processors do not have a dedicated comparator.

3.1. FIFO length

Figure 7 shows the FIFO_A structure in detail. It comprises two FIFOs named FIFO_A0 and FIFO_A1, and a selector. The "write_enable" signal for each FIFO determines which FIFO the input data will be stored into. In addition, the selector controls which output from the FIFOs should be fed to the PE array. Here, the length of FIFO_A0 should be *n*, if we want to handle an $n \times n$ matrix. FIFO_A1 is required to have a length that is sufficient to store n^2/b data if we have a $b \times b$ PE array. For example, in the case of solving a 12×12 matrix problem with a 4×4 PE array, the lengths of FIFO_A0 and FIFO_A1 must be *12* and *36*, respectively. Furthermore, both FIFO_B and FIFO_C are required to have the same length as FIFO_A1. Thus, for the given example, their lengths must be equal to or greater than *36*.

3.2. Procedure

The processing mechanism is described using a simple example of a multiplication of two 12×12 matrices with a 4×4 PE array. At first, all input matrix data are set to appropriate FIFOs. As the size of the input matrix is greater than the size of the PE array, each input matrix is divided to fit the PE-array size, i.e., 4×4. Figure 8 shows the manner in which the divided partial input matrices A and B should be set up in FIFO_A and FIFO_B. In the figure, A_{XX} represents a 4×4 partial matrix. For FIFO_A, all partial matrices of matrix A are initially saved in FIFO_A1. In this example, FIFO_C, which corresponds to matrix C, is used to store the result of the operation. Thus, FIFO_C is initially set to 0. After setting up the initial data in the FIFOs, the matrix-matrix multiplication starts. The data $A_{00}A_{01}A_{02}$ from FIFO_A1 and $B_{00}B_{10}B_{20}$ from FIFO_B are fed to the PE array simultaneously. The "compute" and "data transfer or roll" operations in this mechanism progress in a manner similar to those observed in the processing of the systolic array. Because of the torus connections, the outputs of the PE array are restored to the FIFOs. After *12* compute-and-roll steps, the partial matrix C_{00} is obtained. The data $B_{00}B_{10}B_{20}$ that were fed to the PE array are output from the upper side of the array and restored into FIFO_B through the torus connection. On the other hand, the data $A_{00}A_{01}A_{02}$ fed to FIFO_A is now stored into FIFO_A0, instead of FIFO_A1. An image of the state of FIFO_A just after the abovementioned operations is shown in Fig. 9. Next, for the C_{01} calculation, $A_{00}A_{01}A_{02}$ in FIFO_A0 and $B_{01}B_{11}B_{21}$ in FIFO_B are fed to the PE array. In this step, the data sequences $A_{00}A_{01}A_{02}$ and $B_{01}B_{11}B_{21}$ that were returned from the array are buffered again to FIFO_A0 and FIFO_B, respectively. Here, the order of the data in FIFO_A is the same as that shown in Fig. 9. Next, $A_{00}A_{01}A_{02}$ from FIFO_A0 and $B_{02}B_{12}B_{22}$ from FIFO_B are fed to the PE array, and C_{02} is obtained. As demonstrated above, the data sequence $A_{00}A_{01}A_{02}$ is fed to the array three times, while the data sequence in FIFO_B is fed to the array once. In general, we have to feed *n/b* partial matrix elements in matrix A to the PE array *n/b* times to obtain the *n/b* row partial results of the $n \times n$ final matrix C. Similarly, to obtain the partial matrix results $C_{10}C_{11}C_{12}$ and $C_{20}C_{21}C_{22}$, we have to iterate the set of the abovementioned operation sequences two more times.

Here, the advantage of our architecture is that we can eliminate the external complicated data loading mechanism that is often required in systolic or other similar arrays just by providing some FIFOs and controlling them systematically. This simple I/O data management mechanism is useful when we solve the APP described in Section 2. For example, to solve the APP, we have to feed the entire input matrix to the PE array three times as the APP solving algorithm shown in Fig. 1 has a three-time iteration loop. However, by managing the input matrix data in the same manner as in the case of the matrix-matrix multiplication demonstrated above, all the required data are smoothly fed to the PE array without any delay and skew arrangement.

A set of data is required to be fed from FIFO_C to the PE array only once while the data for FIFO_A and FIFO_B need to be fed *n/b* times. Bearing this fact in mind, reg_c is constructed with two registers reg_c0 and reg_c1, as mentioned previously, and it is shown in Fig. 6. The roles of reg_c0 and reg_c1 change alternatively, and these registers, therefore, form a "double buffer." In other words, while one register is involved in a processing such as $c = c + a * b$ accumulation (where a, b, and c are the elements of matrices A, B, and C, respectively), the other register is used to load new data, i.e., an element of matrix C. When a set of the processing sequence is executed, the role of the registers is reversed. With this mechanism, the loading time of new data can be overlapped with the processing time and therefore hidden in the total processing time.

3.3. Instruction set

As explained previously, with regard to the control unit, we have two types of instructions: micro-instructions and macro instructions. The former is used to directly drive each PE. The latter is formed as a counter field before the micro-instruction. The counter field is used to indicate how many times the attached micro-instruction should be issued. When a macro instruction is fetched at the controller, the content of the counter field is loaded to the counter while the micro-instruction is set to the micro-instruction register in Fig. 5. There are only three basic micro-instructions: "Load," "Move," and "FOU" for immediate data loading, data transfer between the neighboring PEs, and the FOU operation along with the data transfer between the neighboring PEs, respectively. Thus, it is relatively easy to control our array processor if an assembly language is provided.

4. Evaluation and remarks

To verify the feasibility of the proposed architecture, we performed its RTL design with Verilog-HDL, and evaluated it by using an RTL simulator. The actual evaluation was performed using a matrix-matrix multiplication, all-pairs shortest path problem, i.e., one of the APPs, and a 2D discrete cosine transform (DCT). Several input matrices with different sizes were applied to each of the problems. The algorithm was validated by comparing the results with the outputs obtained from software programs that realized the above algorithm and developed without any hardware image. In all evaluation cases, the comparisons were matched 100% and thus, the feasibility of our architecture was confirmed.

Figure 10 shows the circuit size of each PE with/without *QueueDoRegsiter* mechanism. The x-axis represents the matrix size ratio of the input matrix and the PE array (*n/b*), and y-axis represents the number of transistors. Here, all data are 16-bit integer, and the number of transistors is estimated using the outputs of Xilinx ISE 9.1i. The PE circuit without *QueueDoRegsiter* mechanism rapidly increases when the *n/b* ratio becomes large, while that of the PE with the mechanism is constant (about 10k transistors). This is because that the PE featuring *QueueDoRegister* does not need to provide a register file in it to store all input matrix data; the data are stored in the external FIFOs. From the implementation point of view, this fact brings us a great merit that many PEs can be realized on a single silicon die.

5. Conclusions

In this paper, we have proposed a new array processor architecture utilizing FIFOs. In this architecture, by rotating all input data and partial results in the PE array and FIFOs during processing, the required data in each PE are fed on time. Thus, the previously proposed algorithms for array processors such as the APP-solving algorithm, which often requires large data storage in each PE if the size of the problem is large, can be applied without any modifications. In addition, from the structure point of view, our architecture separates the data storage part from the PE array part and realizes it as external FIFOs. This means that a large-scale array processor based on our architecture can be implemented with conventional memory or FIFO chips if we develop a VLSI that realizes the PE array part. In addition, we could realize many PEs on one VLSI chip because each PE is very simplified and consists of only a 3-input fused operation unit and several registers. The VLSI design is one of our future tasks.

Acknowledgements

We would like to thank Prof. Kenichi Kuroda and Prof. Yuichi Okuyama (The University of Aizu) for their helpful discussions and suggestions.

References

[1] S. Y. Kung, “VLSI Array Processors,” Prentice Hall, 1988.

[2] G. Fox, S. Otto, and A. Hey, “Matrix Algorithms on a Hypercube in Matrix Multiplication,” Parallel Computing, Vol. 4, pp.17–31, 1987.

[3] H. T. Kung and C. E. Leiserson, “Systolic Arrays for VLSI,” in Introduction to VLSI Systems, C. A. Mead and L. A. Conway. Reading, MA:Addison-Wesley, 1980, Sec. 8.3.

[4] D.J. Lehmann, “Algebraic structures for transitive closure,” Theoretical Computer Science, Vol. 4, pp. 59–764, 1977.

[5] G. Rote, “Path problems in graphs,” Springer Computing Supplementum, Vol. 7, pp. 155-189, 1990.

[6] F. J. Nunez and M. Valero, “A Block Algorithm for the Algebraic Path Problem and its Execution on a Systolic Array,” Proc. the International Conference on Systolic Arrays, pp.265-274, 1988.

[7] G. Griem and L. Oliker, “Transitive closure on the Imagine stream processor,” Proc. the 5th Workshop on Media and Stream Processors, 2003.

[8] G. Venkatarman, S. Sahni, and S. Mukhopadhyaya, “A blocked all-pairs shortest-paths algorithm,” ACM Journal of Experimental Algorithms, Vol. 8, p. 2.2 , 2003.

[9] L.E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State Univ., Bozeman, MT, 1969.

[10] L.J. Guibas, H.T. Kung, and C.D. Thompson, “Direct VLSI implementation of combinatorial algorithms,” Proc. Conference on VLSI: Architecture, Design, Fabrication, CalTech, pp. 509-525, Jan. 1979.

[11] G. Rote, "A systolic array algorithm for the algebraic path problem (Shortest Paths; Matrix Inversion)" *Computing*, 34, pp.191–219, 1985.

[12] F. J. Nunez and M. Valero, “A Block Algorithm for the Algebraic Path Problem and its Execution on a Systolic Array,” Proc. the International Conference on Systolic Arrays, pp.265-274, 1988.

```
for $pass = 1:3$
 for $step = 0 : b-1$
  for all $[0 \le i, j, k < b]$ & $[(k-j-i) \bmod b = step]$
  begin
   $c_{out} \leftarrow c_{in} \oplus a_{in} \otimes a'_{in}$;
   case$(rs)$:
   (11): $a_{out} \leftarrow c_{out}$; $a'_{out} \leftarrow c_{out}$;  // $i=j=k$ black element
   (01): $a_{out} \leftarrow c_{out}$; $a'_{out} \leftarrow a'_{in}$;  // $i \ne j=k$ red element
   (00): $a_{out} \leftarrow a_{in}$; $a'_{out} \leftarrow a'_{in}$; // $(i \ne k)$&$(j \ne k)$ white element
   (10): $a_{out} \leftarrow a_{in}$; $a'_{out} \leftarrow c_{out}$;  // $i=k \ne j$ blue element
  end
```

Figure 1. APP solving procedure

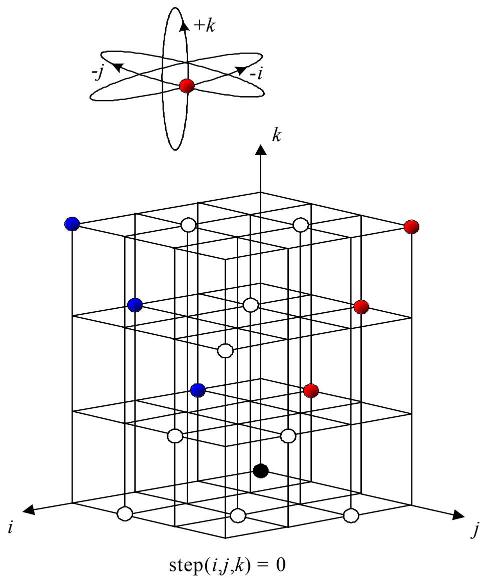

Figure 2. 3D toroidal index space to solve APP

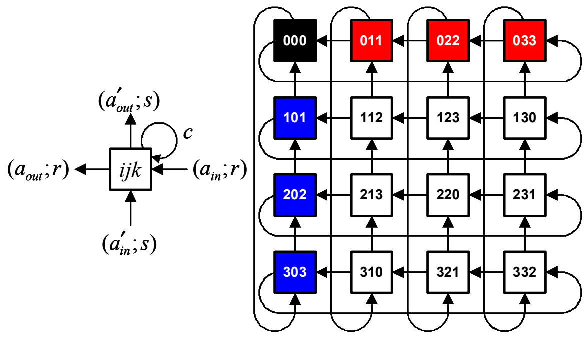

Figure 3. 2D array processor obtained by k-axis toward projection of the 3D toroidal index space shown in Fig. 2

Table 1. Algebraic path problem

S	$\oplus$	$\otimes$	$(*)$	$\bar{0}$	$\bar{1}$	Problem
$\boldsymbol{R}$	+	$\times$	$a^* = 1/(1-a)$	0	1	Matrix Inversion $(I_n - A)^{-1}$
$\{0,1\}$	$\vee$	$\wedge$	$a^* = 1$	0	1	Transitive Closure
$\boldsymbol{R} \cup \{+\infty\}$	min	+	$a^* = 0$	$+\infty$	0	All-pairs Shortest Paths
$\boldsymbol{R} \cup \{-\infty\}$	max	+	$a^* = 0$	$-\infty$	0	All-pairs Longest Paths
$\boldsymbol{R} \cup \{+\infty\}$	max	min	$a^* = \infty$	0	$+\infty$	All-pairs Maximum Capacity Paths
$\boldsymbol{R}_{[0,1]}$	max	$\times$	$a^* = 1$	0	1	All-pairs Maximum Reliability Paths
$\boldsymbol{R} \cup \{+\infty\}$	min	max	$a^* = 0$	$+\infty$	0	Minimum Cost Spanning Tree

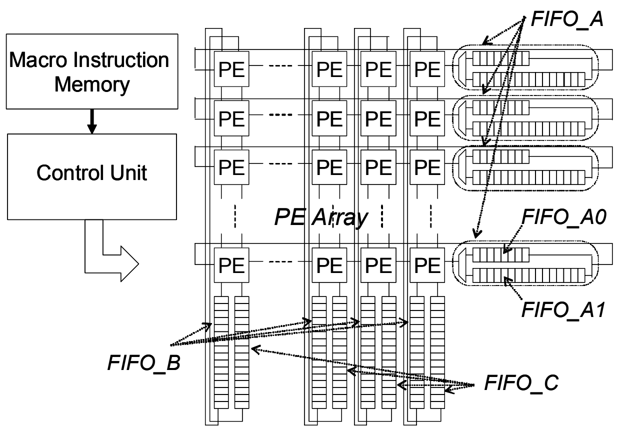

Figure 4. An overview of 2D array processor featuring *QueueDoRegister* mechanism

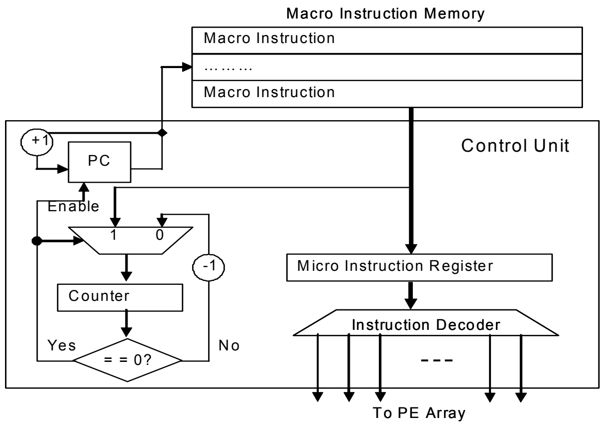

Figure 5. Controller overview

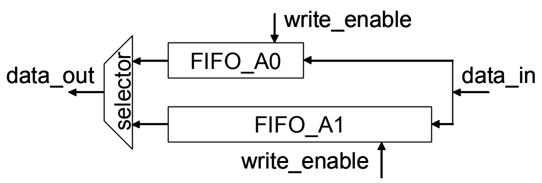

Figure 7. Structure of FIFO_A

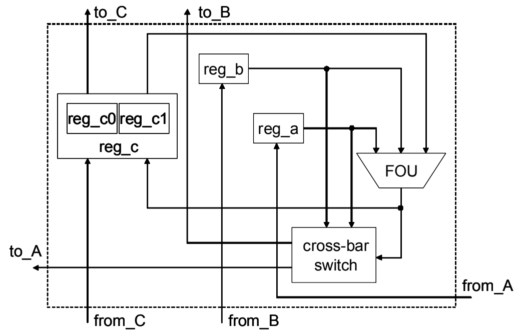

Figure 6. Processing element

$$A = \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{00} & B_{01} & B_{02} \\ B_{10} & B_{11} & B_{12} \\ B_{20} & B_{21} & B_{22} \end{pmatrix}$$

FIFO_A $\cdots A_{00} A_{01} A_{02} A_{10} A_{11} A_{12} A_{20} A_{21} A_{22}$

FIFO_B $\cdots B_{00} B_{10} B_{20} B_{01} B_{11} B_{21} B_{02} B_{12} B_{22}$

Figure 8. Data set-up examples of matrices A and B for FIFO_A and FIFO_B, respectively, when the matrices are blocked

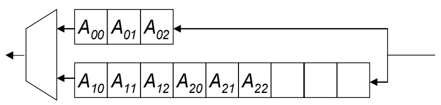

Figure 9. An image of the state of FIFO_A just after the first iteration of a block algorithm is finished

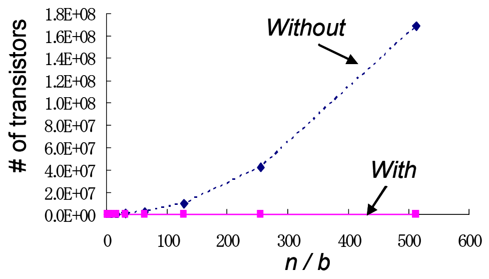

Figure 10. Circuit size of one PE with/without *QueueDoRegister* mechanism