Defining Processing Elements in Dependence Graphs from for-do Programming Constructs.

STAVROS DOKOUZYIANNIS
Aristotle University of Thessaloniki
Department of Electrical and
Computer Engineering
Egnatia St, 54124
GREECE

ARGIRIS MOKIOS
Aristotle University of Thessaloniki
Department of Electrical and
Computer Engineering
Egnatia St, 54124
GREECE

Abstract: The modeling of processing elements (PEs) in dependence graphs (DGs), defined on 2 and 3 level for-do constructs, is presented. Dependence graphs model algorithms described by computer languages, like Fortran, Pascal, C, and C++ and are developed in cases when the mentioned algorithms are to be implemented in FPGA or embedded hardware, in the form of shift-invariant and systolic processing arrays. The paper is focused on defining the function and the input/output signals of the PEs that are used to build shift-invariant DGs.

Key–Words: Dependence graphs, processing elements, parallel algorithms

1 Introduction

The FPGA and other embedded hardware implementations of computer language algorithms (Fortran, Pascal, C, C++) have significant importance in digital design. They support numerous scientific areas like DSP applications, image processing, digital filtering, cryptography, computer arithmetic [1, 2, 3, 4, 5, 6, 7] and many others.

Presently, the automated FPGA/embedded hardware implementation is feasible through VHDL frameworks (ALTERA, Xilinx and other platforms). The programming constructs defined on VHDL are quite circuit compatible, so that the compilation of language structures is based on available equivalent models and structures. In oppose to VHDL, and although loops in algorithms have been studied for decades [8], there is not an available procedure which could compile a computer program algorithm onto a complete DG model. Thus, the existing transformations relate to very regular algorithms, like matrix multiplication, convolution or signal processing applications [9, 10, 11], where the definition of DGs is a result of intuitive work, and it is usually available for low orders of complexity. The regularity of the DG structure is characterized by the term shift-invariance [12] and is the basis for defining iterative constructs by the DG format.

The systolic array [13] is one of the computational models which is implementable on FPGA devices, because it allows the modeling of algorithms by a set of regularly situated processing elements (consisting of computer arithmetic circuits as well as FSMs), interconnected by edges demonstrating the interrelations between them.

This paper concentrates on defining PEs in DGs, i.e., the input/output signals and the performed logical functions, based on a graphical and easily understandable approach. The presentation is limited to 2 and 3-level for-do programming language constructs, since it is only possible to illustrate 2 and $3-Dimensional$ DGs on the Euclidean space.

In Section 2 the multiplication of a matrix by a vector is demonstrated, along with the proposed PE modeling methodology. In Section 3, an extension to a matrix by matrix multiplication is presented. The later multiplication example provides a symmetric placement of the PEs on the DG and is understated as a model for the derivation of PE design of higher order for-do constructs in the near future (section 4). Section 5 summarizes the conclusions.

2 Building processing elements of a Matrix-Vector multiplication.

Initially, the case of a matrix-vector multiplication, i.e., $c = A \cdot b$, is studied. A and b are an $n \times n$ matrix and an $n \times 1$ vector respectively. The elements of vector c are calculated from (1),

$$c_i = \sum_{j=0}^{n-1} a_{ij} \cdot b_j \qquad (1)$$

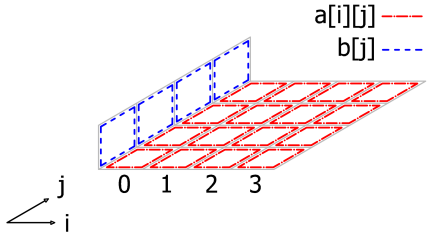


Figure 1: Initial state where all values of matrix A and vector b are presented. The red colored dotted dashed line contours depict elements $a[i][j] : i, j = 0, 1, 2, 3$. The blue colored dashed line contours depict elements $b[j], j = 0, 1, 2, 3$.

where $i = 0, 1, \ldots, n-1$ and the terms a_{ij} and b_j correspond to the elements of matrix A and vector b. Equation (1) can be realized in a computer system by the sequential code that follows:

1: **for** $i = 0,\ i < n-1$ **do**
2: **for** $j = 0,\ j < n-1$ **do**
3: $c[i] = c[i] + a[i][j] \cdot b[j];$
4: **end for**
5: **end for**

When this code segment is executed on a computer system, line 3 is evaluated starting from the right hand side, where the product $a[i][j] \cdot b[j]$ is calculated and then added to the current value of $c[i]$. This value is stored as the next value of $c[i]$. The execution order of the computations is controlled by the system compiler, imposing a software control method. Since this mechanism is unavailable in processor arrays implementations, specific hardware design techniques have to be proposed to realize this behavior.

The transformation of this 2-level for-do loop onto a shift-invariant DG will be realized by examining line by line the corresponding programming code. Fig. 1 is used to illustrate the analysis that is going to follow, where matrix A and vector b are represented by red colored dotted-dashed line and blue colored dashed line contours, respectively.

The execution of line 3 is examined initially, for $i = 0$ and $j = 0, 1, 2, 3$, i.e.:

1: $c[0] = c[0] + a[0][0] \cdot b[0];$
2: $c[0] = c[0] + a[0][1] \cdot b[1];$
3: $c[0] = c[0] + a[0][2] \cdot b[2];$
4: $c[0] = c[0] + a[0][3] \cdot b[3];$

which is illustrated in Fig. 2. During the first step the values of $a[0][0]$ and $b[0]$ are multiplied, thus on Fig. 2a only the corresponding cells are emphasized. The fact that these cells intersect is interpreted as their ability to interact, i.e., perform the multiplication $a[0][0] \cdot b[0]$. In a similar fashion, Fig. 2b, 2c, and 2d illustrate the derived partial products $a[0][1] \cdot b[1]$, $a[0][2] \cdot b[2]$, and $a[0][3] \cdot b[3]$ respectively. In order to calculate value $c[0]$ all the previously generated partial products have to be added together. This action is represented by the insertion of index j in variable c, thus line 3 of the initial algorithm becomes:

$c[i][j] = c[i][j-1] + a[i][j] \cdot b[j];$

This code is considered as a single assignment code, since every variable is assigned one value only, during the execution of the algorithm. The modified code is graphically presented in Fig. 3. The front black coloured solid line contour, in Fig. 3a, defines the accumulated partial product of the previous step that is used to produce the current result, corresponding to the rear black coloured solid line contour. Again, the fact that these cells intersect, is interpreted as their ability to interact, i.e., perform the required addition besides the previously mentioned multiplication. It is noted that in the first step there is no previous partial product, i.e. $c[0][-1] = 0$. In a similar fashion Fig. 2b, 2c, and 2d illustrate the derived accumulated partial products $c[0][1]$, $c[0][2]$, and $c[0][3]$.

The performed computations are visualized, introducing a *virtual cube*. Each side of the cube that is illustrated with a distinct contour, corresponds to a value that is inserted or extracted from it, according to the direction of the calculations defined by the indices ordering. If the dimensions of matrix A were set to 4×4 and of vector b to 4×1 then $c[0][3]$ would be the first element of the new vector c.

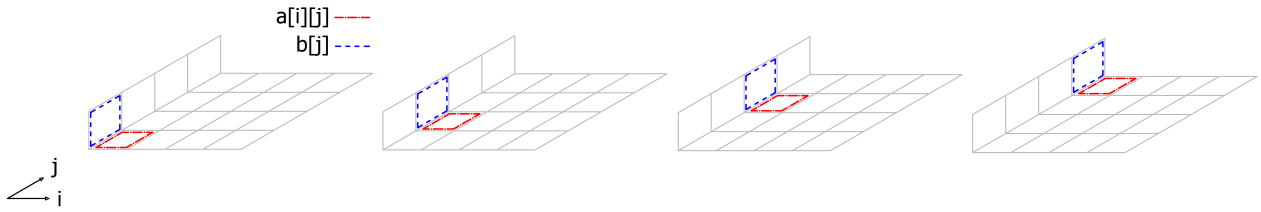
Analyzing further the algorithm, line 3 is executed for $i = 1$ and $j = 0, 1, 2, 3$. During the first step the values of $a[1][0]$ and $b[0]$ are multiplied. The corresponding cells don't intersect, thus not all the values required to perform the multiplication are available locally. Specifically, value $b[0]$ has to be broadcasted to the current cube where value $a[1][0]$ resides.

2.1 Converting program code to recursive form.

In order to make the necessary values locally available, index i is inserted in variable b and the matrix vector multiplication algorithm is transformed into:

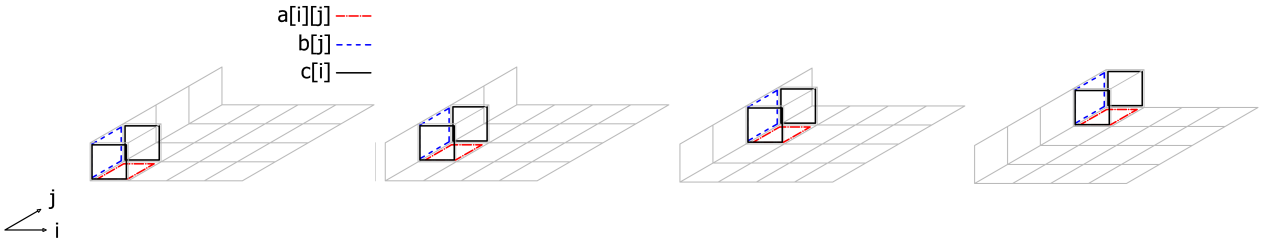
1: **for** $i = 0,\ i < n-1$ **do**
2: **for** $j = 0,\ j < n-1$ **do**
3: $b[i][j] = b[i-1][j];$
4: $c[i][j] = c[i][j-1] + a[i][j] \cdot b[i][j];$
5: **end for**
6: **end for**

The algorithm is now in a *localized form* since all the variables are, only, directly dependent upon the variables of their neighboring node. Declaring index j as the recursion index, the algorithm may be considered as a localized recursive algorithm. In Fig. 4



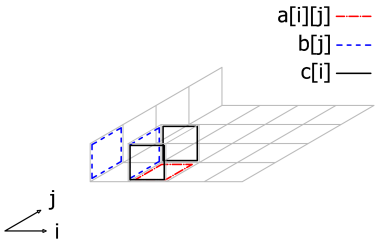
(a) $a[0][0]$, $b[0]$ (b) $a[0][1]$, $b[1]$ (c) $a[0][2]$, $b[2]$ (d) $a[0][3]$, $b[3]$

Figure 2: Execution of the code for values $i = 0$ and $j = 0, 1, 2, 3$

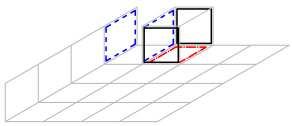


(a) $a[0][0]$, $b[0]$, $c[0][-1]$, $c[0][0]$ (b) $a[0][1]$, $b[1]$, $c[0][0]$, $c[0][1]$ (c) $a[0][2]$, $b[2]$, $c[0][1]$, $c[0][2]$ (d) $a[0][3]$, $b[3]$, $c[0][2]$, $c[0][3]$

Figure 3: Insertion of index j in variable c



(a) $a[1][0]$, $b[0]$, $c[1][-1]$, $c[1][0]$



(b) $a[1][3]$, $b[3]$, $c[1][2]$, $c[1][3]$

Figure 4: Insertion of index j in variable b

the insertion of index i to variable b is depicted by the rightmost blue coloured dashed line contour that represents the transmission of value $b[0]$ to the next cube. Since the cells intersect, the required computations are performed and vector c is generated.

2.2 Determination of the processing elements.

The virtual cubes, introduced so far to the matrix-vector multiplication example, form the basis for the PE design, in general. The transformation requires 4 steps:

Step 1: Every virtual cube generates its corresponding PE in the DG.

Step 2: The computations realized in each virtual cube determine the function assigned to the respective PE.

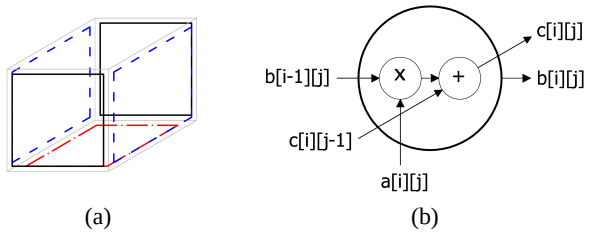
Step 3: For any two neighboring PEs, an interconnection arc is generated, if the adjacent sides of their respective virtual cubes have the same line style colored contour. The direction of the generated arc is defined by the chronological sequence of the computations.

Step 4: The initial surfaces become the primary input signals of the DG.

Fig. 5 represents the resulting PE, derived according to the proposed method, while Fig. 6 presents the resulting DG.

3 Matrix-Matrix multiplication

Having already studied the derivation of a DG for a matrix-vector product algorithm, the case of a two $n \times n$ matrices, A and B, multiplication algorithm can also be examined. The formula that calculates the



(a) (b)

Figure 5: (a) Virtual cube, and (b) the resulting processing element (PE).

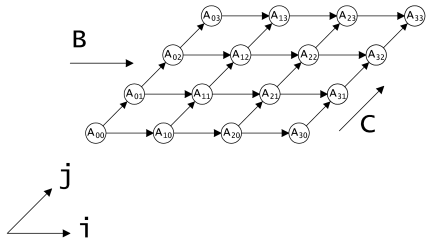


Figure 6: Dependence graph modeling the shift-invariant multiplication of a 4×4 matrix and a 4×1 vector.

elements of the product is given in (2).

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \tag{2}$$

Terms a_{ik} and b_{kj} represent the elements of matrices A and B. From (2), the following sequential code is derived:

1: **for** $i = 0,\ i < n-1$ **do**
2: **for** $j = 0,\ j < n-1$ **do**
3: **for** $k = 0,\ k < n-1$ **do**
4: $c[i][j] = c[i][j] + a[i][k] \cdot b[k][j]$
5: **end for**
6: **end for**
7: **end for**

The resulting surfaces of matrices A and B, as well as, their constitutive elements are presented in Fig. 7.

3.1 Converting the code to recursive form.

Working as in the previous example, the algorithm is initially transformed in a single assignment form, inserting index k to variable c. Moreover, indices j and i are inserted to variables a and b, respectively, in order to overcome the burden of broadcasting a_{ik} to all cubes with the same i and k coordinates, and b_{kj} to all cubes with the same k and j. Considering k as the recursion index, the algorithm obtains the following locally recursive form:

1: **for** $i = 0,\ i < n-1$ **do**

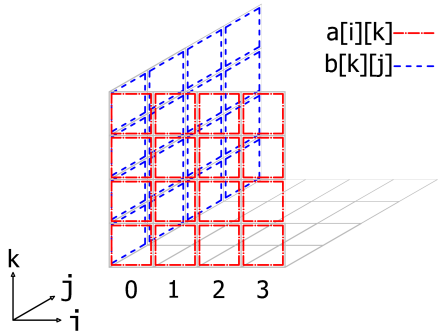


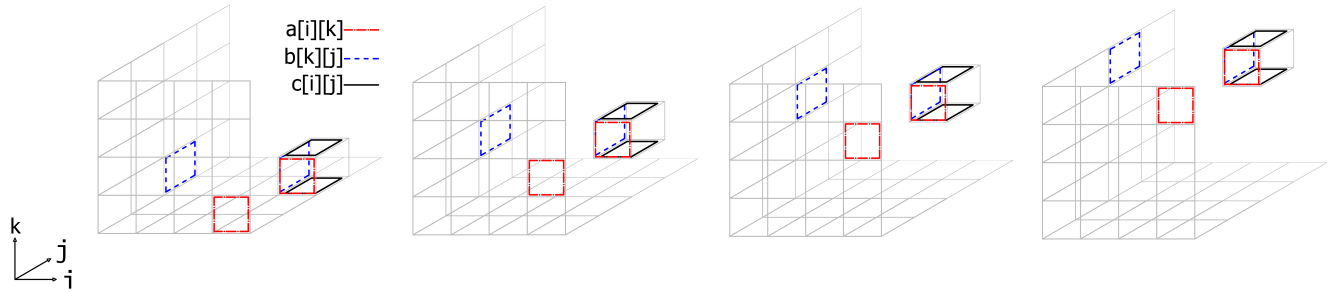
Figure 7: Initial state where all values of matrix A and B are presented. The red colored dotted dashed line contours depict elements $a[i][k] : i, k = 0, 1, 2, 3$. The blue colored dashed line contours depict elements $b[k][j], k, j = 0, 1, 2, 3$.

2: **for** $j = 0,\ j < n-1$ **do**
3: **for** $k = 0,\ k < n-1$ **do**
4: $a[i][j][k] = a[i][j-1][k]$
5: $b[i][j][k] = b[i-1][j][k]$
6: $c[i][j][k] = c[i][j][k-1] + a[i][j][k] \cdot b[i][j][k]$
7: **end for**
8: **end for**
9: **end for**

Examining the execution of the algorithm for $i = 3$, $j = 2$ and $k = 0, 1, 2, 3$ the snapshots of Fig. 8 are created. The values of matrix A and B are transmitted from cube to cube as they are being used for the computation of the partial products. This behavior is represented by the red colored dotted-dashed and blue colored dashed line contours that are repeated throughout the design. In the same fashion the initial partial product generated every time $j = 0$, is transmitted to the next cube in order to generate the next accumulated partial product. Thus, every cube except the first one, receives the accumulated partial product and transmits the one that it produces. This is depicted by the bottom and top black colored solid line contours of Fig. 8. The cells with $k = n-1$ produce the final results.

3.2 Processing elements and dependence graph derivation.

Given the virtual cube of Fig. 9a, the resulting PE is given in 9b, after following the methodology proposed in Section 2.2. The DG that corresponds to the product of two 4×4 matrices, is presented on Fig. 10.



(a) $a[3][0]$, $a[0][2]$, $a[3][2][-1]$, $a[3][2][0]$ (b) $a[3][1]$, $a[1][2]$, $a[3][2][0]$, $a[3][2][1]$ (c) $a[3][2]$, $a[2][2]$, $a[3][2][1]$, $a[3][2][2]$ (d) $a[3][3]$, $a[3][2]$, $a[3][2][2]$, $a[3][2][3]$

Figure 8: Execution of the code for values $i = 3$, $j = 2$ and $k = 0, 1, 2, 3$

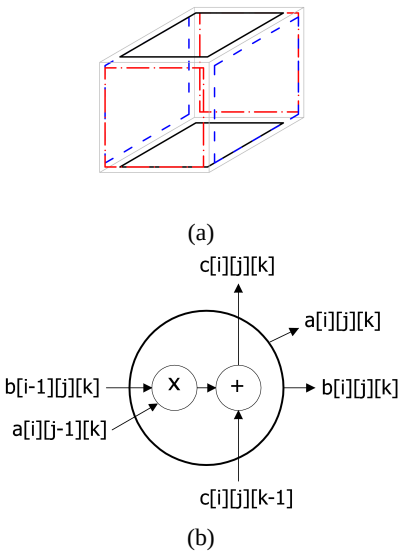


Figure 9: (a) Virtual cube and (b) the resulting processing element (PE).

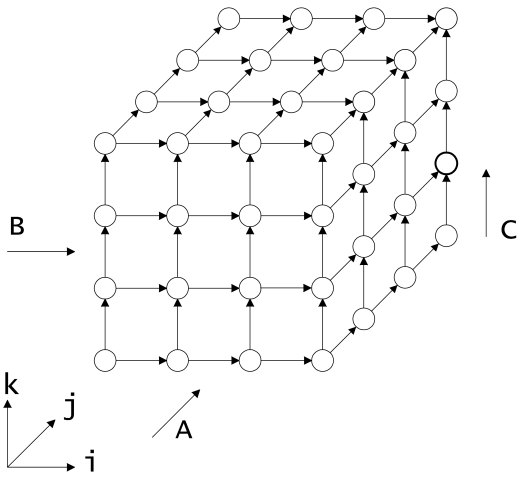


Figure 10: Dependence graph modeling the shift-invariant multiplication of two 4×4 matrices.

4 Higher-level for-do programming constructs.

The case of higher-level constructs have to be further examined. Instead of graphical modeling steps, as it was successfully performed in the previous two examples, which have given optically understandable results, the for-do loops of order higher than 3 can not be illustratively presented on the Euclidean space through DG modeling. Formal methods have to be developed in the future, which will provide automated generation of DGs. Moreover, the mentioned formal methods seem to be able to exploit the experience of the steps involved in this work. Namely,

a. the conversion of a for-do construct to a recursive form,

b. the formation of a virtual cube for determining the input-output signals and the function performed by the PEs under construction.

This modeling for 2 and 3-order for-do loops, provide some optimism for a successful transformation approach of higher order loops.

5 Conclusions

The method presented in this work relates to the well known problem of designing systolic array processors and is targeted on their implementation on FPGA and other embedded hardware. It considers the transformation of for-do constructs in algorithms described by computer programs (Fortran, Pascal, C, C++ and others), onto DGs, which model graphically these programs; and it is particularly focused on the analytic extraction of PEs.

The formulation of PEs is being performed in an understandable, graphical way, using a 2-$Dimensional$ matrix by vector multiplication example and a 3-$Dimensional$ matrix by matrix multiplication example. The demonstrated clarity of the PE

formulation is accompanied by the simplicity of defining its input/output signals and the processing function, which in a general case can be a complex FSM machine.

The significance of the developed method is that is sets the basis for the development of formal methods, in the near future, which will allow the formulation of PEs for higher order (3-and more) for-do programming constructs. Up to now these constructs can not be demonstrated in 3 dimensions and their formulation is based on complex intuitive approaches and human imagination.

References:

[1] C. Castro-Pareja, J. Jagadeesh, S. Venugopal, and R. Shekhar, "FPGA-based 3d median filtering using word-parallel systolic arrays," in *Proc. International Symposium on Circuits and Systems, ISCAS '04*, vol. 3, May23-26 2004, pp. 157–160.

[2] A. Lopich and P. Dudek, "Architecture of a VLSI cellular processor array for synchronous/asynchronous image processing," in *Proc. International Symposium on Circuits and Systems, ISCAS '06*, May21-24 2006, pp. CD–ROM.

[3] M. D. Ercegovac and J.-M. Muller, "Arithmetic processor for solving tridiagonal systems of linear equations," in *Fortieth Asilomar Conference on Signals, Systems and Computers, 2006. ACSSC '06.*, vol. 3, Oct.Nov.29-1 2006, pp. 337–340.

[4] O. Nibouche, M. Nibouche, and A. Bouridane, "High speed FPGA implementation of RSA encryption algorithm," p. 204207, Dec.14-17 2003.

[5] A. Amira, A. Bouridane, P. Milligan, and M. Roula, "An FPGA implementation of walsh-hadamard transforms for signal processing," in *Proc. 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing, (ICASSP '01)*, vol. 2, May7-11 2001, pp. 1105–1108.

[6] A. Amira and A. Bouridane, "An FPGA implementation of discrete hartley transforms," in *Proc. Seventh International Symposium on Signal Processing and Its Applications*, vol. 1, Jul.1-4 2003, pp. 625–628.

[7] A. Amira, A. Bouridane, P. Milligan, and A. Belatreche, "Design of efficient architectures for discrete orthogonal transforms using bit level systolic structures," in *Proc. IEEE Computers and Digital Techniques*, vol. 149, no. 1, Jan. 2002, pp. 17–24.

[8] L. Lamport, "The parallel execution of do loops," *Communications of the ACM*, vol. 17, pp. 83–93, Feb. 1974.

[9] R. Karp, R. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, Jul. 1967.

[10] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *Proc. IEEE of 11th Annual Symposium on Computer Architecture*, 1984, pp. 208–214.

[11] S. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. dissertation, Stanford University, Information Systems Laboratory, 1985.

[12] S.-Y. Kung, *VLSI Arrays Processors*. Englewood Cliffs, NJ 07632: Prentice Hall, 1987.

[13] H. Kung and C. Leiserson, "Systolic arrays for VLSI," in *Sparse Matrix Proceedings, Philadelphia, PA: Society of Industrial and Applied Mathematicians*, 1978, pp. 245–282.