A Technique for Mapping Sparse Matrix Computations into Regular Processor Arrays

Roman Wyrzykowski[1] and Juri Kanevski[2]

[1] Dept. of Math. & Comp. Sci, Czestochowa Technical University, Dabrowskiego 73, 42-200 Czestochowa, Poland
[2] Dept. of Electronics, Technical University of Koszalin, Partyzantow 17, 75-411 Koszalin, Poland

Abstract. A technique for mapping irregular sparse matrix computations into regular parallel networks is proposed. It is based on regularization of the original irregular graph of an algorithm. For this aim, we use a mapping of an original index space corresponding to dense matrices into a new one, which corresponds to a chosen sparse-matrix storage scheme. This regularization is followed by space-time mappings, which transform the algorithm graph into resulting networks. The proposed approach is illustrated by the example of mapping matrix-vector multiplications.

1 Introduction

At present, there exist [1, 7, 10, 12] efficient methods for mapping regular computations into application-specific processor arrays. These arrays are [5, 7] VLSI-oriented regular processor architectures with primarily local interconnections between processing elements (PEs). Using the existing mapping methods, efficient array architectures for solving linear algebraic operations over dense matrices have been designed [5, 7, 11].

In practice, however, matrices with zeros as their prevailing elements (or sparse matrices) are used very often (e.g., in solving finite element problems [2]). Using this feature of input/output data, it is possible to improve the efficiency of large matrix computations radically [8]. However, the existing mapping methods can be used only for sparse matrices with a special structure (e.g. banded matrices), not allowing to deal with sparse matrices with a more general sparsity structure, which leads to irregular computations [6].

In the paper, we propose a technique for extending the capabilities of the existing mapping methods on irregular matrix computations. It is based on regularization of the original irregular graph of a given algorithm. For this aim, we use a mapping of an original index space corresponding to dense matrices into a new index space which corresponds to a chosen sparse-matrix storage scheme. This regularization is followed by applying properly the well-known technique [5] of space-time mappings, which transform the algorithm graph into resulting architectures. The proposed approach allows us to derive regular processor arrays (PAs) with primarily local interconnections, providing parallel implementations of such important sparse matrix problems as, for example, matrix-matrix and matrix-vector multiplications, LU decomposition and solving linear systems.

2 Mapping Regular Computations

PAs can be designed systematically by applying linear (or affine) mappings to algorithms that are expressed by systems of recursive equations or nested loops [5]. Basically, the design is composed of the following three components: an algorithm specifying the computation; an allocation mapping that maps computations to PEs; a schedule mapping specifying the execution time for each computation in the algorithm.

Nested loops with regular dependencies can be represented [12] by regular or quasi-regular dependence graphs (DGs), or a composition of them. Each node of such a DG corresponds to a certain operator (or iteration) of the original algorithm, and is associated with an integer vector $\mathbf{K} = (k_1, ..., k_n)'$; all the nodes are located in vertices $\mathbf{K}$ of a lattice $K^n \subset \mathbf{Z}^n$, where K^n is called the index space. Arcs between nodes of this DG (or dependencies between operators of the algorithm) are represented by a dependence matrix $\mathbf{D}$, in which the i-th column is a dependence vector $\mathbf{d}_i$. For a strictly regular DG, these vectors are independent of $\mathbf{K} \in K^n$. For a quasi-regular DG, the matrix $\mathbf{D}$ splits into a regular $\mathbf{D}^*$ and nonregular $\mathbf{D}^{**}$ submatrices. For strictly regular DGs, $\mathbf{D} = \mathbf{D}^*$.

Definition 1 [12]. A structural scheme $\mathbf{C}$ of a processor array implementing the given algorithm $\mathbf{AL}$ with the DG $\mathbf{G}$ is a 3-tuple $\mathbf{C} =< S, T, \Phi >$, where $S = < V_S, E_S >$ is a directed graph called the array structure, T is the synchronization function specifying the computation time of nodes in the DG, and Φ is the set of PE operation algorithms.

One of the most promising approaches to mapping recursive algorithms with regular dependencies into PAs consists in [10, 12, 13] finding first the set of all possible and nonequivalent allocation mappings $\mathbf{F}_S(\mathbf{K})$ satisfying given constraints for links between PEs, which are located in vertices of a lattice $K^m \subset \mathbf{Z}^m$. For each of network topologies S corresponding to this set, an optimal schedule mapping which implements the algorithm correctly (i.e. preserving all data dependencies without conflicts) is find then. This mapping is constructed as a linear (or affine) function $\mathbf{F}_T$ with n unknown coefficients.

3 Mapping Sparse Matrix Computations

3.1 Storage Schemes for Sparse Matrices

For sparse matrices, it is a common practice to store only the nonzero elements with information about their locations in a matrix [8]. A variety of storage schemes are used [4, 8, 9] to store and process sparse matrices. These specialized schemes not only save storage, but also yield computational savings because unnecessary multiplication and additions with zero can be avoided. There is no single best data structure for storing sparse matrices [4].

Since computation overhead increases with increasing the complexity of sparse -matrix storage schemes, it seems reasonable to limit oneself to rather simple schemes (or formats) when implementing sparse matrix computation on processor arrays. These schemes are as follows [4, 9]:

1. **compressed sparse row** (CSR) or **compressed sparse column** (CSC) format, which uses one real and two integer arrays to store an $n \times n$ sparse matrix $\mathbf{A}$ with q nonzero elements;
2. **rowwise ITPACK/ELLPACK** (or shortly **I/E**) format, which uses the following two arrays: (a) an $n \times j_{max}$ real array A_S contains the nonzero elements of the corresponding row of the sparse matrix $\mathbf{A}$, where j_{max} is the maximum number of nonzeros in any row of $\mathbf{A}$; (b) an $n \times j_{max}$ integer array JA stores the column numbers of the corresponding entries in A_S;
3. **columnwise ITPACK/ELLPACK** format;
4. **diagonal storage** format;
5. **jagged – diagonal** format;

The application range of scheme 5 is limited [9] to regularly stuctured matrices consisting of a few diagonals. In the case of parallel implementation, the last scheme needs [6] a complicated scheme of asynchronous synchronization to be involved. That is why, we leave only schemes 1-4 for the further consideration.

3.2 Regularization of Sparse Matrix Computations

In a resulting DG, which corresponds to the transition to one of the above-described storage schemes, some variables of the original algorithm will be propagated between nodes of the DG in a nonregular and nonlocal way. This is illustrated in Fig.1b, where the DG of a sparse matrix-vector multiplication of the form $\mathbf{Ax} = \mathbf{y}$ is shown. Here we assume that the input $N \times N$ matrix $\mathbf{A}$ is represented (see Fig.1a) in the columnwise I/E format. In the DG, the input variable x is propagated in a strictly local and regular way, which is described by the dependence vector $\mathbf{d}_X = (1,0)^t$. However, the generation of the output variable y is carried out in a fully nonregular and nonlocal way. Consequently, this generation should be regularized and localized.

We will distinguish the following three forms (or ways) of regularization (and thereby localization) of propagating indexed variables of an algorithm between nodes of its DG.

Definition 2. The first way of regularization of propagating a certain indexed variable of an algorithm consists in introducing a minimally necessary amount of redundant nodes into the original DG, in order that the resulting propagation is described by a fixed vector $\mathbf{d}$ of the regular component $\mathbf{D}^*$ of the dependence matrix $\mathbf{D}$.

For linear algebraic algorithms, the introduction of redundant nodes can be interpreted as a process of appearing some redundant operations with zero.

Definition 3. The second way of regularization consists in choosing such an allocation mapping $\mathbf{F}_S$ that all the transfers of a certain indexed variable of an algorithm are carried out within fixed PEs, i.e. satisfying the following condition:

$$\mathbf{F}_S\,\mathbf{d} = 0 \qquad (1)$$

where $\mathbf{d} = \mathbf{K}_2 - \mathbf{K}_1$ is a dependence vector describing the propagation of the variable between nodes of the DG.

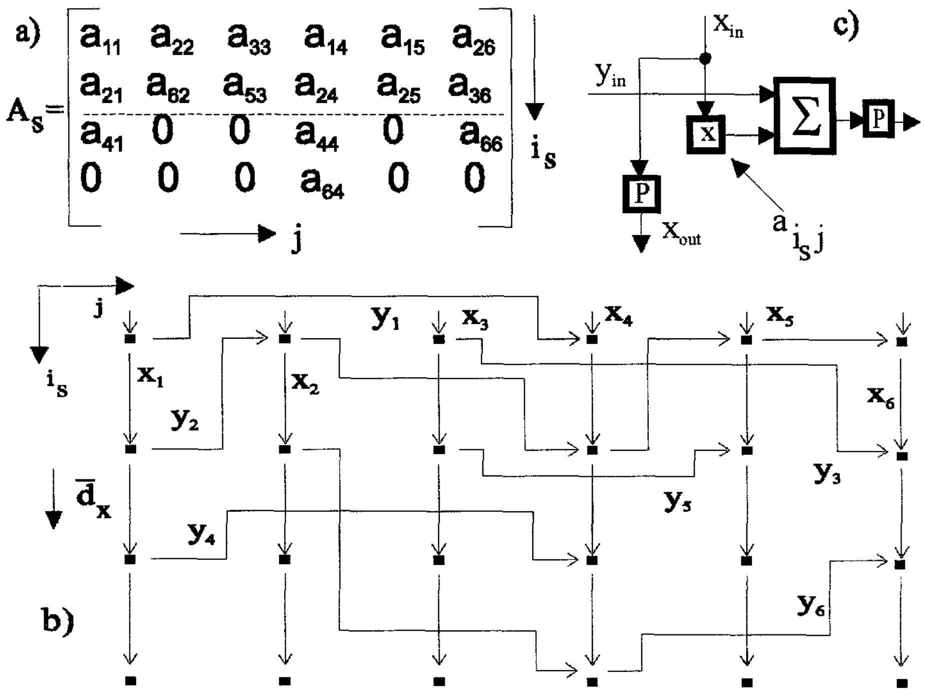

Fig. 1. Synthesis of PAs for sparse matrix-vector multiplications, using the columnwise I/E format: (a) sparse representation of original matrix; (b) original DG; (c) its node

Definition 4. The third way of regularization of propagating a certain input variable consists in preloading all the elements of the corresponding input data arrays into local memories of PEs, from where these elements are fetch when necessary. In the case of an output variable, the intermediate values of all the entries in the corresponding data arrays are computed within fixed PEs, from where these entries are unloaded using their pipelining propagation between PEs; this propagation also enables generating the final values of these entries.

Like the first form, the third one also needs to introduce some additional nodes into the DG of an algorithm. These nodes together with new arcs, which are responsible for the pipelining propagation of variables between PEs, provide I/O of these variables. At the same time, the process of either fetching elements of the input data arrays from local memories or computing intermediate values of entries in the output arrays within fixed PEs will be described by vectors **d** satisfying condition (1). Therefore, the second form, unlike the rest of them, is oriented on regularizing not the DG of an algorithm, but the resultant graph giving an array structure S. Moreover, for all these forms, it is not necessary to localize variable transfers between nodes of the DG. This localization should be provided only on the level of structures S.

3.3 Mapping Procedure

Let us assume that a numeric algorithm corresponds to processing sparse matrices represented in one of the storage schemes 1-4 chosen in Section 3.1. Based on the above-defined forms of graph regularization, and the previously proposed [10, 12, 13] methods for the synthesis of allocation/schedule mappings $\{\mathbf{F}_S, \mathbf{F}_T\}$, the following procedure for deriving the set of permissible structural schemes $\mathbf{C} =< S, T, \Phi >$ of PAs implementing the algorithm is formulated:

1. Based on the basic DG $\mathbf{G}_B$ of the algorithm, divide all the indexed variables of the algorithm into the following two groups: (a) the first group contains variables characterized by their regular propagation between nodes of the DG; (b) the rest of indexed variables are included into the second group.
2. From dependence vectors $\mathbf{d}$ describing the propgation of variables belonging to the first group, create a regular component $\mathbf{D}^*$ of the matrix $\mathbf{D}$.
3. Using the method proposed in work [13], determine the set of all the permissible allocation mappings $\mathbf{F}_S$, where any mapping must satisfy the locality condition for those interprocessor links which correspond to the matrix $\mathbf{D}^*$. For each permissible mapping $\mathbf{F}_S$, perform steps 4-7.
4. Determine a partial structure S^* corresponding to the mapping $\mathbf{F}_S$ and component $\mathbf{D}^*$.
5. From the second group, exclude variables not satisfying condition (1).
6. Using the first and third ways of regularization, perform the regularization for the remaining variables of the second group.
7. For a regularized DG $\mathbf{G}_R$ obtained in this way, determine first the resulting structure S which corresponds to $\mathbf{F}_S$. Then using, for example, the method proposed in work [10], find an optimal schedule mapping $\mathbf{F}_T$ (and thereby the synchronization function T). Finally, based on the graph $\mathbf{G}_R$ and the couple $\{\mathbf{F}_S, \mathbf{F}_T\}$, find a set Φ of operation algorithms of PEs.

The proposed procedure is specified in the constructive proofs of Theorems 1 and 2. These theorems deal with so called coordinate DGs whose dependence matrices $\mathbf{D}$ are given by the identity matrices $\mathbf{I}_n$, where $n = 2, 3$. This case is very important from the practical point of view because it enables us to cope with such important sparse matrix problems as, for example, matrix-matrix and matrix-vector multiplications, LU decomposition and solving linear systems.

Theorem 1. *Let us assume that in the case of processing dense matrices, a numeric algorithm is described by a 2-D coordinate DG with* $\mathbf{D} = \mathbf{I}_2$. *If this algorithm is determined over sparse matrices represented in the columnwise or rowwise I/E format, then the use of the proposed mapping procedure enables us to obtain an 1-D processor array with* $O(i_{max})$ *or* $O(j_{max})$ *PEs, respectively, where* i_{max} *or* j_{max} *is the maximum number of nonzeros in any column or row of the input matrix* $\mathbf{A}$*, respectively.*

Proof. Let an input matrix $\mathbf{A}$ is represented in the columnwise I/ E format (the proof for the rowwise variant is similar). In this case, nodes of the basic graph $\mathbf{G}_B$, which corresponds to processing the matrix $\mathbf{A}_S$, will be located in vertices of

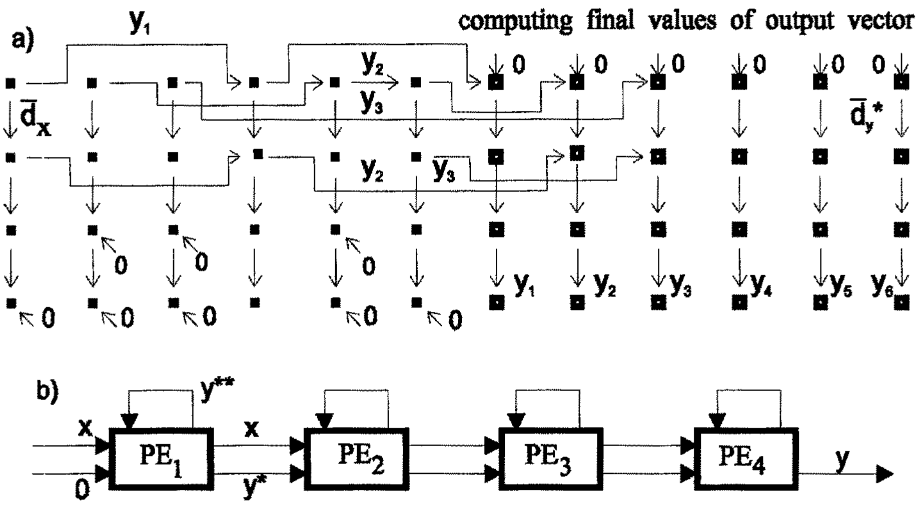

Fig. 2. Synthesis of PAs for sparse matrix-vector multiplications, using the columnwise I/E format: (a) DG after regularization; (b) resulting array structure.

the integer lattice $K^2 = \{\mathbf{K} = (i_S, j) : 1 \le i_S \le v_S = O(i_{max});\ 1 \le j \le O(N)\}$. Moreover, the mapping procedure proposed above turns into the following one:

1. In the graph $\mathbf{G}_B$, there are two independent indexed variables. The fully local propagation of the first variable (along columns of $\mathbf{A}_S$) is given by $\mathbf{d}_1 = (1,0)^t$, while the second variable is propagated in a fully nonregular and nonlocal way, along rows of $\mathbf{A}_S$ (see Fig.1c). Consequently, $\mathbf{D}^* = [\mathbf{d}_1]$.
2. After completing the locality condition (constructed for links corresponding to $\mathbf{D}^*$) by the requirements of minimizing the number of PEs, a single permissible allocation mapping given by $\mathbf{F}_S = [1, 0]$ is obtained.
3. The partial structure S^* consists of v_S PEs, which are connected through a single unidirectional channel for propagating the first variable.
4. The propagation of the second indexed variable is regularized in the third way. For the 2-D coordinate DG, such a regularization is always possible. As a result, some new arcs given by vectors $\mathbf{d}^* = \mathbf{d}_1$ appear in the DG (see Fig.2a). They are responsible for the pipelining propagation of the second variable between PEs. Vectors $\mathbf{d}^{**} = (0, d_j)^t$, which satisfy constraint (1), correspond to the "circulation" of the second variable within fixed PEs.
5. The resulting structure S corresponding to the graph $\mathbf{G}_R$ obtained after regularization, as before contains $v_S = O(j_{max})$ PEs. They are connected through an additional unidirectional channel for either input (see Fig.2a) or output of the second variable. Based on the matrix $\mathbf{D} = [\mathbf{d}_1\ \mathbf{d}^{**}]$, the schedule mapping given by $\mathbf{F}_T = [1\ \ 1]$ is derived as giving the minimum total execution time of the algorithm. Since $\Delta_T = \mathbf{F}_T\ \mathbf{D} = [1\ \delta^{**}]$, the both variables of the algorithm are transferred between PEs with the time-delay of one cycle.

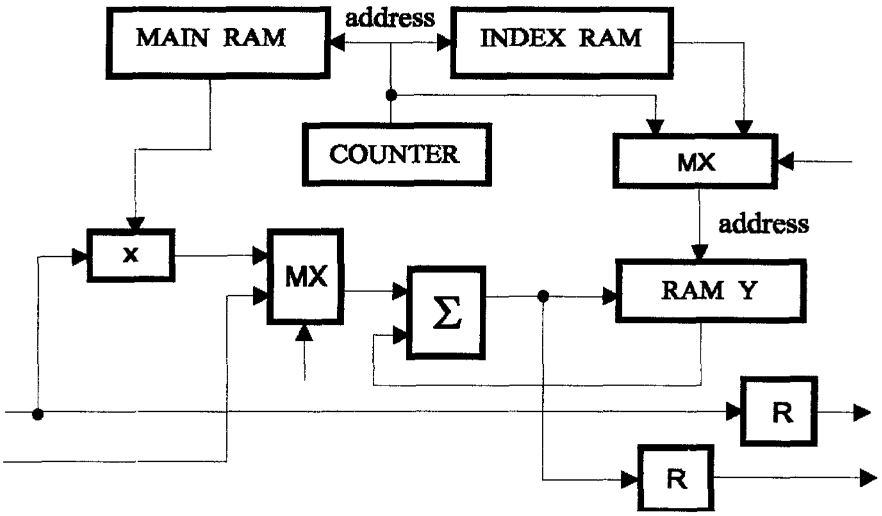

Fig. 3. Internal structure of PEs shown in Fig.2 (R, RAM, M, Σ or x stands for register, random access memory, multiplexer, adder or multiplier, respectively).

Theorem 2. *Let us assume that in the case of processing dense matrices, an original numeric algorithm is described by a 3-D coordinate DG with* $\mathbf{D} = \mathbf{I}_3$. *If this algorithm is determined over sparse matrices represented in the compressed sparse row CSR or column CSC format, then the use of the proposed mapping procedure enables us to obtain 1-D processor arrays containing* $O(N_p)$ *PEs and providing the execution of the algorithm in* $T = k_q + O(N_p)$ *time steps (without taking into account input/output operations), where* N_p *is one of sizes of matrices being processed, and* k_q *is the number of nonzero entries in one of these matrices.*

4 Mapping Sparse Matrix-Vector Multiplications

The design of PAs for sparse matrix-vector multiplications is carried out in accordance with the proof of Theorem 1. Depending on if columnwise or rowwise I/E format is used, a couple of dual arrays are derived. The structures S for the first array is shown in Fig.2b, while the internal structures of PEs in this array is presented in Fig.3.

To provide the implementation of the algorithm on a fixed number $K < i_{max}, j_{max}$ of PEs, a decomposition of the matrix $\mathbf{A}_S$ can be applied. This decomposition consists in splitting $\mathbf{A}_S$ into either $s =]i_{max}/K[$ horizontal strips (see Fig.1b) or $s =]j_{max}/K[$ vertical strips, which are processed sequentially. Such an approach also allows us to decrease the number of redundant nodes considerably. In fact, assuming the columnwise I/E format, for the jth column of $\mathbf{A}_S$ it becomes possible to avoid executing those redundant operations which correspond to zero entries of $\mathbf{A}_S$ with row indices i_S satisfying the following inequality: $i_S > K*]N_j/K[$, where N_j is the number of nonzeros in the jth column of $\mathbf{A}$. As a result, for the columnwise format, the algorithm will be executed on K PEs in $T_M = \sum_{j=1}^{N}]N_j/K[+ (N+K)$ time steps. Here the first component corresponds to the accumulation of intermediate values of elements of the vector $\mathbf{y}$ within fixed PEs, while the second one corresponds to computing and unloading final values of these elements, with the duration of any step determined by time required for a scalar multiplication.

In many applications, for example, when solving linear systems by an iterative method, a number of matrix-vector multiplications is performed sequentially. In this mode, the asymptotic execution time T_M^a for a single multiplication (or block pipelining period [5]) can be made less than T_M. For the columnwise format, such a decrease results from the possibility of overlapping the computation of final values of entries in the output vector $\mathbf{y}$ for a current iteration with the accumulation of intermediate values of entries in the output vector for the next iteration. As a result, we have $T_M^a = \sum_{j=1}^{N}]N_J/K[+ K$ time steps.

This overlapping requires a relatively small hardware overhead. Only RAM $\mathbf{y}$ and adder, or RAM $\mathbf{x}$ are duplicated for the columnwise or rowwise format, respectively. At this cost, the processor utilization η_M^a is improved considerably:

$$\eta_M^a = k_a/(T_M^a\ K) = k_A/(k_A + K^2)$$

Here k_A is the number of nonzero entries in $\mathbf{A}$. Assuming that $k_A >> K^2$, we obtain $\eta_M^a \approx 1$. This value confirms the high efficiency of the proposed parallel architectures when implementing iterative computations.

References

1. Darte, A., Robert, Y.: Mapping uniform loop nests onto distributed memory architectures. Parallel Computing **20** (1994) 679-710
2. Hammond, S.W., Law, K.H.: Architecture and operation of a systolic engine for finite element computations. Computers and Structures **30** (1988) 365-374
3. Jennings, A., McKeown, J.J.: *Matrix computation.* J. Willey & Sons, 1992
4. Kumar, V., Grama, A., Gupta, A., Karypis, G.: *Introduction to parallel computing.* Benjamin/Cummings Publish. Comp., 1994
5. Kung,.Y.: *VLSI array processors.* Prentice-Hall, Englewood Cliffs, 1988
6. Melhem, R.: Solution of linear systems with striped sparse matrices. Parallel Comput. **6** (1988) 165-184
7. Moreno, J.H., Lang, T.: *Matrix computations on systolic-type arrays.* Kluwer, 1992
8. Pissanetzky, Z.: *Sparse matrix technology.* Academic Press, London, 1984
9. Saad, Y.: Krylov subspace methods on supercomputers. SIAM J. Sci. Stat. Comput. **10** (1989) 1200-1232
10. Shang, W., Fortes, J.A.B.: On time mapping of uniform dependence algorithms into lower dimensional processor arrays. IEEE Trans. Parallel and Distr. Systems **3** (1992) 350-363
11. Wyrzykowski, R.: Processor arrays for matrix triangularisation with partial pivoting. IEE Proc. E, Comput. Digit. Tech. **139** (1992) 165-169
12. Wyrzykowski, R., Kanevski, J., Maslennikov, O.: Mapping recursive algorithms into processor arrays, in *Proc. Int. Workshop Parallel Numerics'94*, M. Vajtersic and P. Zinterhof eds., Bratislava, 1994, 169-191
13. Zhong, X., Rajopadhye, S., Wong, I.: Systematic generation of linear allocation functions in systolic array design. J. VLSI Signal Processing **4** (1992), 279-293