Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

ARRAYS FOR PARTITIONED MATRIX ALGORITHMS: TRADEOFFS BETWEEN CELL STORAGE AND CELL BANDWIDTH

Jaime H. Moreno
Tomas Lang

August 1988
CSD-880065

Arrays for partitioned matrix algorithms: tradeoffs between cell storage and cell bandwidth

Jaime H. Moreno and Tomás Lang *

Computer Science Department
University of California Los Angeles
3680 Boelter Hall, Los Angeles, Calif. 90024

Abstract

A graph-based partitioning method for designing systolic arrays for matrix computations is extended to apply it to processing elements with a small local memory. The introduction of this memory produces a reduction in the cell communication bandwidth and facilitates the use of pipelining within cells. As a consequence, efficient arrays can be designed using the extended method combined with technological parameters that define the ratio between processor speed and communication bandwidth.

The extended partitioning method also allows evaluating tradeoffs between linear and two-dimensional arrays. We illustrate the method using a cube-shaped canonical algorithm, which is communication and computation intensive, and triangularization by Givens' rotations.

1 Introduction

Arrays of processing elements (PEs or cells) have been proposed as an attractive alternative for the implementation of matrix computations. These computations are used for many applications, particularly in signal processing. Specific algorithms have been mapped into arrays and methods have been proposed for performing such mapping in a systematic manner [1].

The *systolic* model of computation has received much attention since its introduction [2] and the field is very active, as can be inferred from recent publications [3]. The popularity of this model has encouraged the use of the term systolic for a variety of arrays, many of which do not use the systolic mode of communication and/or deviate from the original model of simple and numerous cells suitable for VLSI implementation [4,5].

H.T. Kung has recently discussed the impact in performance due to systolic flow of data versus data access to local memory [6]. He concludes that "systolic communications allow cells to compute at a speed higher than what its local memory can support. Systolic communication is a method of increasing the performance of a cell without increasing its local memory bandwidth." This

*J. Moreno has been supported by an IBM Computer Sciences Fellowship. This research has also been supported in part by the Office of Naval Research, Contract N00014-83-K-0493 "Specifications and Design Methodologies for High-Speed Fault-Tolerant Algorithms and Structures for VLSI"

assertion is valid for an array where cells have higher communication bandwidth than local memory bandwidth, which might be the case when local storage is large and external to the cell chip. On the other hand, it seems very appropriate to design an array in which the cell chip contains a limited amount of memory, so that access to this memory is faster than communication among chips. In this case, the fast memory access increases cell performance without requiring fast communication between cells.

In this paper, we consider arrays formed with three different type of cells: systolic (without local memory), pseudo–systolic (with a small amount of memory) and local–access (with an amount of memory $O(n)$, where n is the size of the data), and show the tradeoffs involved between size of memory and communication bandwidth.

We then consider the *partitioning problem* for these arrays, that is mapping large and variable size matrix computations onto an array with fewer cells than the size of the data [7]. We describe a solution that is a generalization of a graph–based method for purely systolic arrays that we have presented previously [8]. We conclude that, with this unified method, it is straightforward to select the appropriate mapping for specific implementation constraints, which determines the ratio between processor speed and communication bandwidth. We show that a small and limited memory in each cell can be easily utilized to reduce cell communication bandwidth and therefore alleviate the communication bottleneck that characterizes systolic arrays. In addition, such small local memory facilitates the use of pipelining within cells. It is also simple to address and manage the local storage, which is not the case with some other methods. The resulting arrays exhibit high utilization, no overhead due to partitioning and simple control.

Moreover, our technique allows evaluating tradeoffs between linear and two–dimensional arrays for partitioned execution of algorithms. We compare properties of linear and two–dimensional structures and conclude that linear arrays offer better performance and implementation than two–dimensional arrays with the same number of cells.

We illustrate the method using first an algorithm that exhibits the highest requirements in terms of operations and communications, and then using triangularization by Givens' rotations, as a real example.

2 Systolic, pseudo–systolic and local–access arrays

To study the tradeoffs between memory and communication bandwidth in arrays for matrix computations, we make the following assumptions regarding algorithms and arrays:

1. Matrix algorithms consist of primitive operations with up to three operands. That is, a matrix algorithm is composed of a sequence of operations that involve up to three operands each. Such an assumption is based on the fact that matrix algorithms of interest consist of arithmetic or logic operations that are unary, binary or ternary at most.
2. All operations have the same computation time. The validity of such an assumption is highly implementation dependent, as suggested by studies about the design of special purpose cells. Moreover, if cells are pipelined, the stage time is the same for all operations.

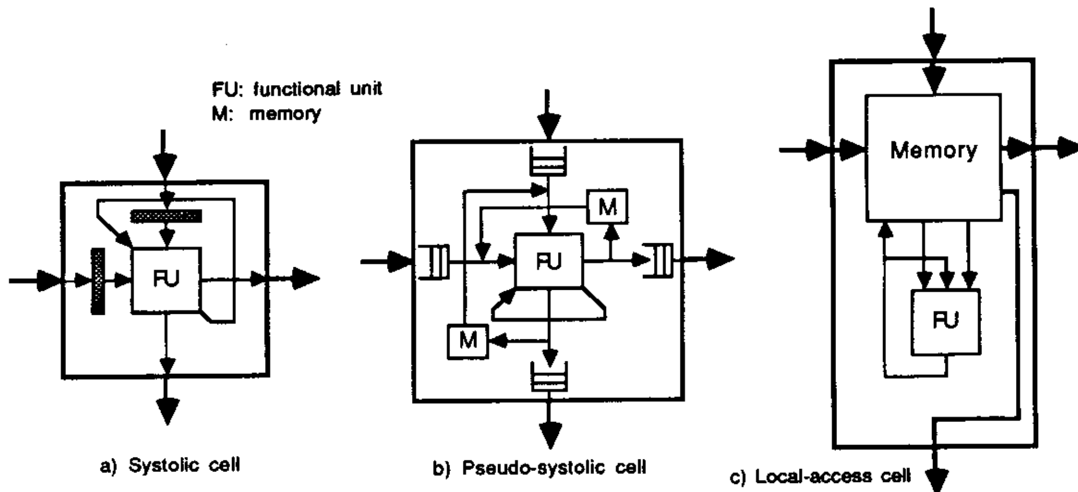


Figure 1: Cells for the different types of arrays

3. Arrays are either linear or two–dimensional structures, as is the case for existing implementations. Moreover, cells of an array have two input and two output ports that allow connecting them in such linear or two–dimensional structures.

4. At every time–step, a cell starts an operation that takes up to three operands and produces up to three outputs. Such outputs can be results computed within the cell or operands that are passed through the cell without modification (i.e., implement broadcasting by transmittent data [9]).

Consequently, the execution of a matrix algorithm in an array requires to specify the flow of up to three data elements per operation and schedule such operations throughout the entire array.

We regard an array as a collection of cells connected in nearest–neighbor fashion, where such cells correspond to one of the following three types:

Systolic cell: *cell with no local storage except for registers used to latch input operands* for an operation. Figure 1a depicts a systolic cell with its corresponding input/output ports. For ternary operations, that is those that require three operands, two operands are received from outside a cell through ports and the third source of data is a feedback loop within the cell. Such a loop can be regarded as a third communications port. For unary and binary operations, only one or two ports are active (i.e., carry data for such operations). Data flows through cells in such a way that *every operation in each cell requires one data transfer per active port.*

Pseudo–systolic cell: *cell with a small and fixed-size memory* (i.e., memory size independent of the size of problems to be solved in the array). Such memory is composed of two separate banks. Figure 1b depicts a pseudo–systolic cell and its ports. Ports and local memory provide two sources of data for every operation. The third source of data is a feedback loop within the cell.

Since the size of local memory is fixed and small, we assume that access time to such local memory matches the functional unit execution rate (i.e., cell pipeline stage time) and that it is shorter than the time to transfer data among cells. This property is exploited by performing

several operations with data from local memory. Consequently, pseudo-systolic cells don't need to receive data through ports at every cycle and *communication bandwidth of pseudo-systolic cells is lower than their computation rate.* Such lower communication rate is adjusted to cell computation rate by FIFO queues attached to ports.

Local-access cell: *cell with memory whose size is proportional to the size of problems to be solved in an array.* Figure 1c depicts a local-access cell. Operations are performed in each cell with up to two operands obtained from local memory, so that data received from neighbor cells is stored in memory before it is used. The third source of data is a feedback loop within the cell.

Local-access cells have large local memory with the objective of storing locally a large portion of data and reduce communications among cells. Consequently, *communication rate among cells is much lower than computation rate* (i.e., much less than one word per port per time-step).

An important conclusion is readily available from the properties of the cells above: *systolic arrays allow maximum parallelism, higher than pseudo-systolic and local-access arrays.* In a systolic cell, each data element is used once and then immediately re-used in the same cell for another operation or passed to another cell (because there is no memory to store data). On the other hand, cells with local memory perform operations with different data on the same cell. Memory in each cell contains elements that are not used at every time step, so that there are elements idle and total parallelism is less than the maximum.*

If the maximum parallelism is not used then arrays with a given number of cells provide the same throughput regardless of the type of cell used as long as cells have the same step-time. The main difference among such arrays is the tradeoff between communication bandwidth and local storage. In one extreme, systolic cells require high communication bandwidth and no local storage. On the other end, local-access cells have a large local memory and low communication rate, while pseudo-systolic cells fall somewhere in between with little local memory. These properties are summarized in Figure 2, where we consider a matrix algorithm that consists of $O(n^3)$ operations (n is the dimension of the matrix). Mapping the algorithm onto an array with K cells leads to the results indicated in the figure (the origin of such values will be presented later). Figure 2 shows that adding local memory to cells reduces communication bandwidth proportionally to the inverse square-root of the size of such a local memory.

Figure 2 also indicates the most suitable implementation for each type of cell, based on communication requirements. For example, systolic cells are better implemented in WSI because such technology provides communication bandwidth between cells of the same magnitude as computation rate (there is no need to go off-wafer). On the other hand, pseudo-systolic cells can be implemented as one cell per chip because such chip could provide high computation rate with data from local memory and lower communication bandwidth between cells (which requires to go off-chip).

An important problem in mapping algorithms to any of the arrays above is *partitioning,* that is decomposing an algorithm so that it can be executed in an array with fewer units than the size

*We assume that no data is duplicated within the array, so that it is not possible to have a local copy of an element that has also been transferred to another cell. As far as we know, this is the case of all arrays proposed in the literature.

	Systolic cell	Pseudo–systolic cell	Local–access cell
Memory per cell	None	$\approx S$	$\approx n^2/K$
Cell communication bandwidth per port [words/time–step]	1	$1/\sqrt{S}$	$\sqrt{K}/n$
Most suitable implementation	WSI	cell per chip (functional unit and small memory)	cell and memory per board

K: number of cells

Figure 2: Tradeoffs between local memory and cell bandwidth

of the data. In the next section, we center our attention on such issue and describe a graph–based partitioning procedure that considers the type of cell used in an array.

3 Graph–based partitioning for systolic, pseudo–systolic and local–access arrays

As discussed in [7], three basic approaches have been proposed to perform partitioning of matrix algorithms: *coalescing, cut-and-pile* and *decomposition into sub-algorithms.* We present now a partitioning method based on the dependency graph of algorithms that uses a combination of coalescing and cut–and–pile, and can be used to derive the three types of arrays described earlier. This method is a generalization of the one given in [8], which used cut–and–pile as the partitioning approach and generated only systolic arrays. As a result, this extended method is capable to trade local memory and cell bandwidth in an implementation, and exploit internal pipelining within cells. We illustrate the technique using the algorithm for triangularization by Givens' rotations shown in Figure 3 [10].

Our partitioning procedure is as follows:

1. Draw the fully–parallel dependency–graph [11] of the algorithm. Such a graph is obtained by tracing the execution of a sequential algorithm (i.e., symbolic execution of the algorithm that tracks which variables are used and when). Figure 4 depicts the fully–parallel dependency–graph for triangularization by Givens' rotations of a 5 by 5 matrix.

2. Transform the fully–parallel dependency–graph into a tri–dimensional graph that we call a *multi–mesh dependency–graph.* To achieve this objective, perform transformations as those indicated in [8] to remove properties not suitable for an implementation. Figure 5 shows the multi–mesh dependency–graph derived from the fully–parallel dependency–graph in Figure 4.

3. Transform the graph obtained in (2) into a new graph, which we call *G–graph,* by *coalescing*

Input: A_{nxn}, B_{nx1}

> **For** r **from** 1 **to** $n-1$
> **begin**
>   **For** i **from** $(r+1)$ **to** n
>   **begin**
> $\theta_{ri} = -\arctan\left(\frac{a_{ir}}{a_{rr}}\right)$, $a_{rr} = \sqrt{a_{ir}^2 + a_{rr}^2}$; Rotation angle
>   **For** j **from** $(r+1)$ **to** n
>   **begin**
> $$\begin{bmatrix} a_{rj} \\ a_{ij} \end{bmatrix} = \begin{bmatrix} \cos\theta_{ri} & -\sin\theta_{ri} \\ \sin\theta_{ri} & \cos\theta_{ri} \end{bmatrix} \begin{bmatrix} a_{rj} \\ a_{ij} \end{bmatrix}$$; Rotation
>   **end**
> $$\begin{bmatrix} b_r \\ b_i \end{bmatrix} = \begin{bmatrix} \cos\theta_{ri} & -\sin\theta_{ri} \\ \sin\theta_{ri} & \cos\theta_{ri} \end{bmatrix} \begin{bmatrix} b_r \\ b_i \end{bmatrix}$$; Rotation
> **end**

Output: U : upper triangular matrix = upper triangular part of rotated A
D : rotated B

Figure 3: Example of a matrix algorithm: Triangularization by Givens' rotations

sets of neighbor primitive nodes of the graph into new nodes (*G–nodes*). In other words, sets of neighbor nodes in the graph are grouped into G–nodes whose functionality and computation time are given by the primitive nodes. An example is shown in Figure 6, where paths along the Z–axis of the graph in Figure 5 have been coalesced into G–nodes.

Criteria to select the sets of nodes composing G–nodes depend on the characteristics of the target array, as discussed later, though the G–graph should be a two–dimensional graph so that it can be mapped onto linear and two–dimensional arrays. Such criteria corresponds to the extension of this method with respect to the one in [8], because of its applicability to the different type of arrays.

4. Divide (i.e., cut)the G–graph obtained in (3) into sets of neighbor G–nodes (*G–sets*). Nodes in a G–set will be executed concurrently in the target array. Consequently, G–sets should have as many G–nodes as there are cells in the array and the dependency structure of such G–nodes should match the communication capabilities of the array. Figure 7 shows an example. For good utilization, all G–nodes in a G–set should have the same computation time (i.e., perform the same number of operations).

5. Schedule (i.e., pile) G–sets for execution in the array, as depicted in Figure 7. G–sets scheduled successively are executed in overlapped (pipelined) manner in the array and data flows among G–sets.

   In [8], we show that data needed to schedule execution of the next G–set is available before the G–set in execution completes. Moreover, outputs from one G–set are used as inputs for G–sets scheduled later so that such outputs need to be stored in memories external to the array, as shown in Figure 7. In the same paper, we also show that due to this scheduling process the array I/O bandwidth can be made identical for linear and two–dimensional arrays

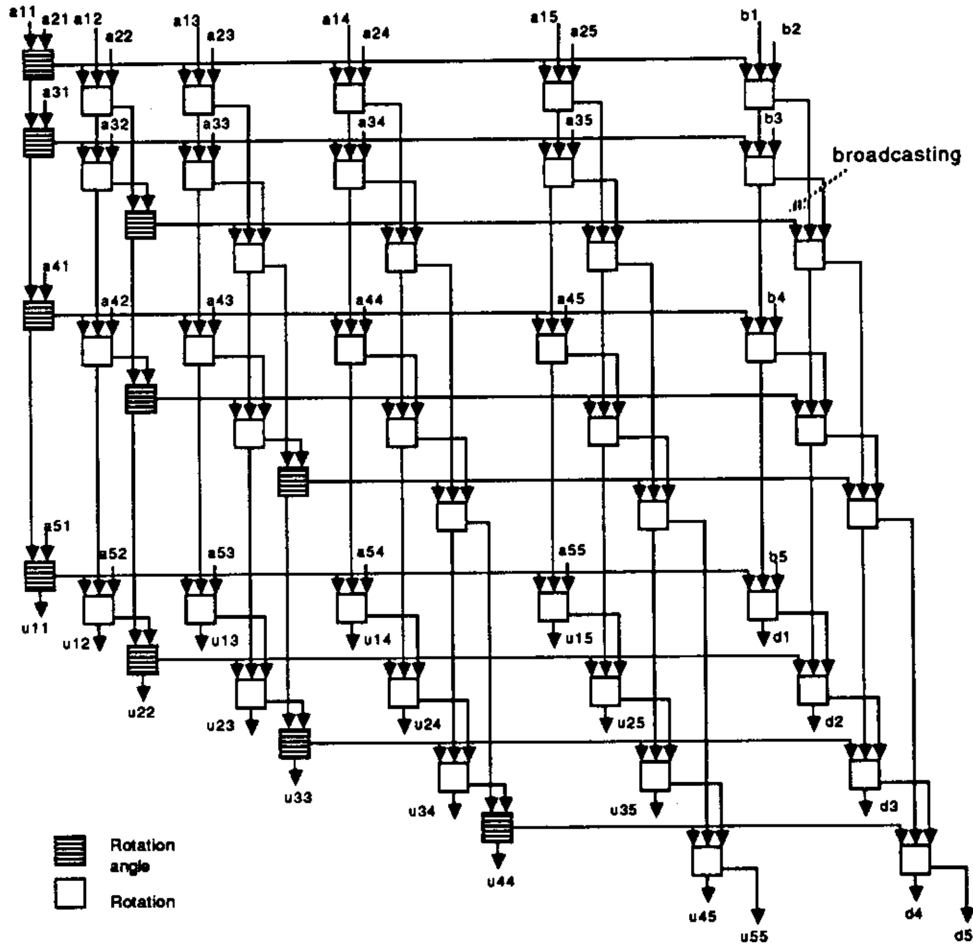


Figure 4: Fully–parallel dependency–graph for triangularization by Givens' rotations

using the I/O structure shown in Figure 7. Such structure consists of a chain of modules (the R blocks in the figure) composed of a register and a memory.

Now that we have described our graph–based partitioning method, we center our attention on the use of this tool to partition algorithms for the three different classes of arrays that we introduced earlier.

4 Partitioning and suitable arrays

The graph–based partitioning method presented in the previous section can be used to derive and evaluate systolic arrays (with external memory), pseudo–systolic arrays (with external memories) and local–access arrays. The selection of one of these architectures as the target for an implementation determines how the multi–mesh dependency–graph is transformed into a G–graph.

For simplicity of exposition, we start our analysis using a multi–mesh dependency–graph that has the cube–shaped structure shown in Figure 8. We call such a graph a *complete multi–mesh*

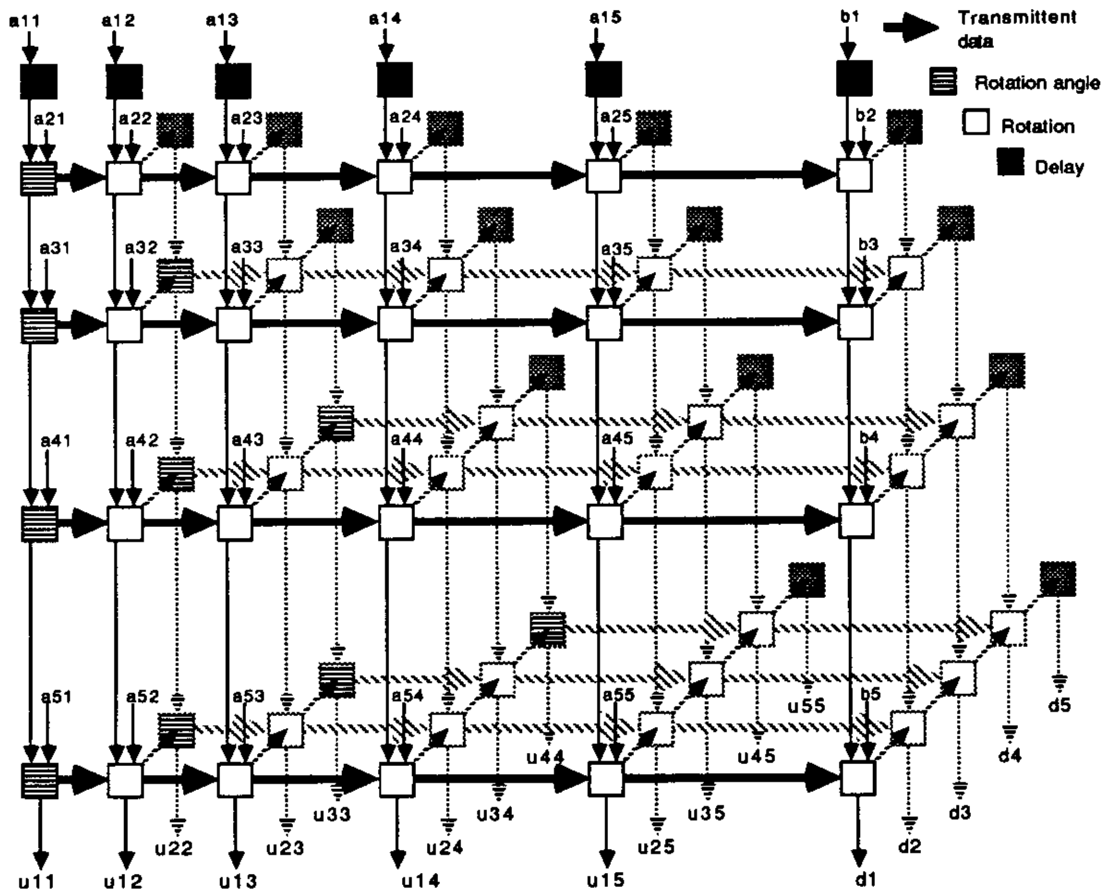


Figure 5: Multi-mesh dependency-graph for triangularization by Givens' rotations

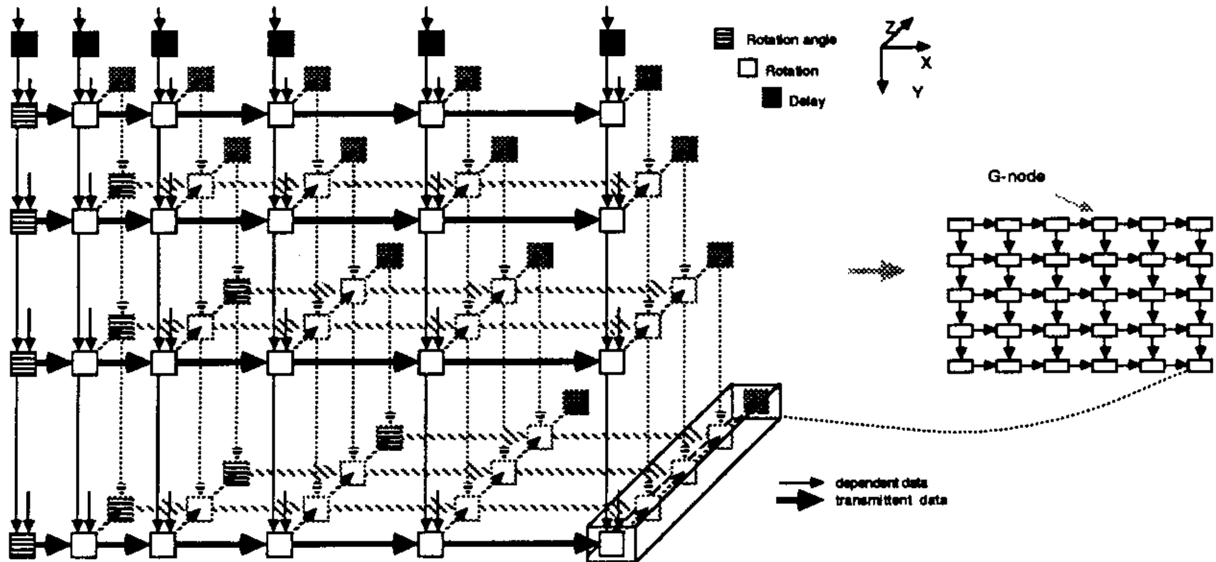


Figure 6: Collapsing primitive nodes into G-nodes

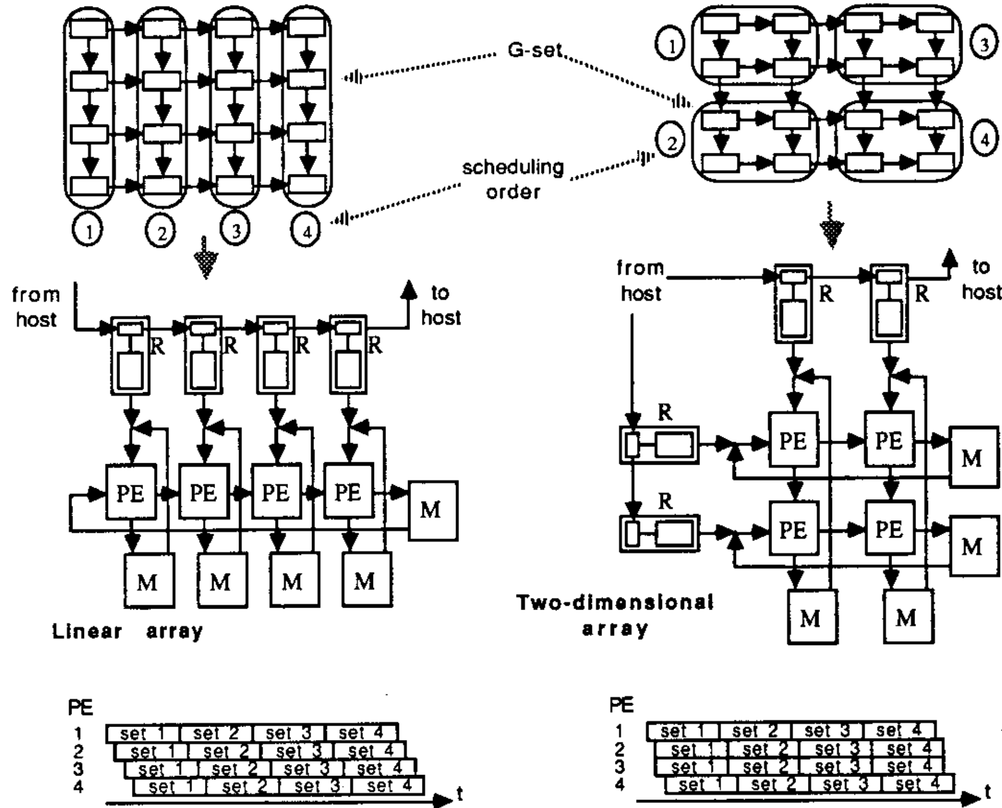


Figure 7: Mapping G–graph into linear and two–dimensional arrays

dependency–graph (CMMDG), because it consists of many rectangular meshes of primitive nodes that are dependent.

A CMMDG corresponds to an algorithm with the most stringent requirements, because it has the maximum number of operations and dependencies that can exist for a given mesh size. Later, we'll address algorithms that have lower computational requirements and consequently are not represented by CMMDGs. However, our method is applied in the same manner in both cases.

4.1 G–graph for a complete multi–mesh dependency–graph

In the context of our graph–based method, partitioning a CMMDG consists of transforming such CMMDG into a G–graph (step 3 of our procedure) and mapping the G–graph onto an array (steps 4 and 5).

To transform a CMMDG into a G–graph, we coalesce sets of neighbor primitive nodes into G–nodes. As stated earlier, criteria to select primitive nodes depends on the target array. Such selection can be regarded as dividing the CMMDG into sub–graphs of p by q by n primitive nodes and coalescing such sub–graphs into G–nodes, as depicted in Figure 9a. These sub–graphs correspond to prisms within the CMMDG. There are three alternatives when deriving such prisms, which correspond to drawing prisms that enclose primitive nodes along each of the axes of the multi–mesh graph. Figure 8 illustrates the case of drawing prisms along axis Z. The resulting

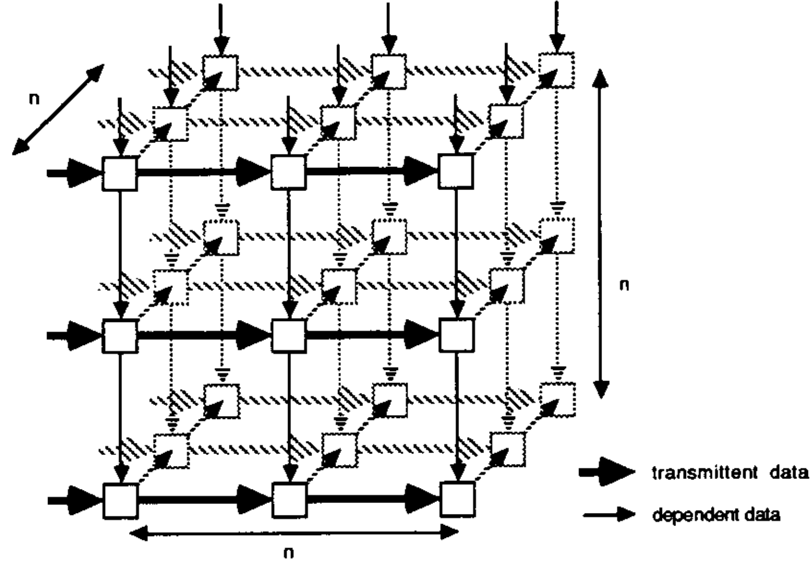


Figure 8: Complete multi–mesh dependency–graph (CMMDG)

G–graph is later divided (cut) into G–sets and such G–sets are scheduled (piled) for execution in an array.

Coalescing into G–nodes determines the following characteristics of the G–graph and of the array:

Number of primitive nodes per G–node = pqn, because the CMMDG is divided into subgraphs (prisms) of size p by q by n primitive nodes. Consequently, the time to execute a G–node is $t_g = pqn$.

Size of G–graph (i.e., number of G–nodes) = (n^2/pq), because there are n^3 primitive nodes that are coalesced into G–nodes of size pqn.

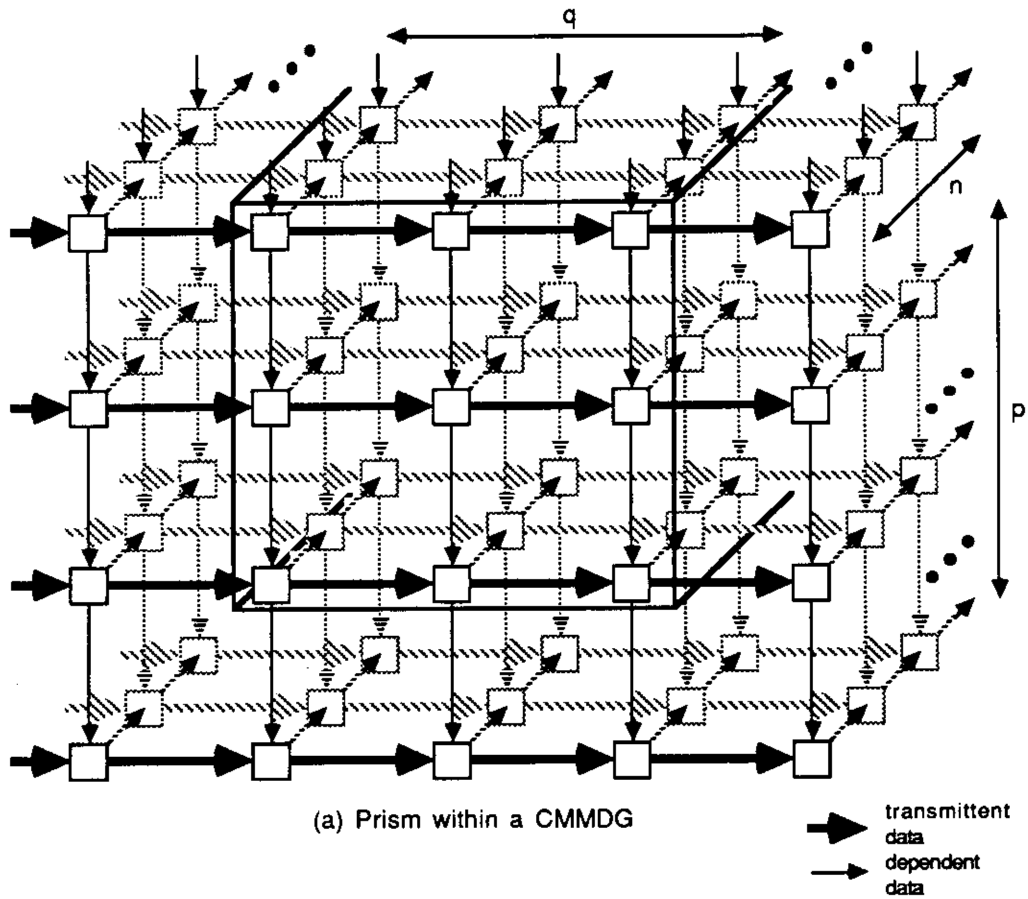
Cell communication bandwidth per port, given by the edges of the CMMDG that are cut by the prisms that define G–nodes. Such edges represent data that arrives or leaves a cell, as shown in Figure 9b. Consequently:

Cell communication bandwidth, horizontal port = $1/q$, because q operations are performed before reaching the rightmost boundary of the prism that defines a G–node.

Cell communication bandwidth, vertical port = $1/p$, because p operations are performed before reaching the lower boundary of the prism that defines a G–node.

Scheduling of primitive nodes within a G–node. This scheduling should be done along meshes composing the CMMDG, because dependencies among nodes correspond to edges of such meshes. Due to storage per cell requirements discussed below, one should choose a scheduling order that traverses meshes of small size. Such ordering corresponds to traversing meshes of size p by q, because p and q are smaller than n. Nodes within such meshes can be scheduled in any order that doesn't violate the dependencies.

Storage per cell. An operation that reads as many data elements from memory as it stores



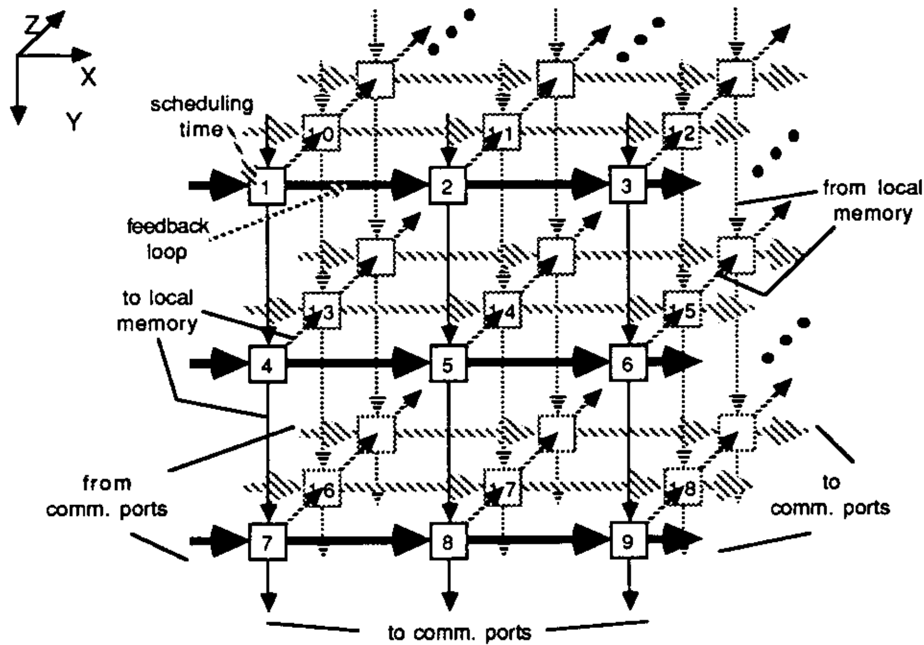


Figure 9: Deriving G–graph for a complete multi–mesh dependency–graph

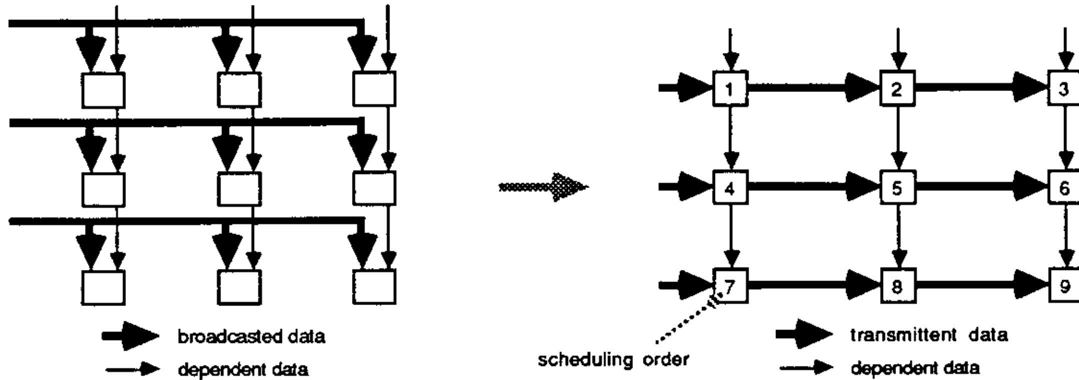


Figure 10: Independent nodes in flow of transmittent data

results in such memory doesn't require additional storage because results can be saved in the same locations used by the operands. On the other hand, an operation that reads fewer operands from memory than it stores results requires as many additional memory words as the difference between read and write operations. Consequently, memory needed to execute a G–node is determined by the maximum number of storage words required in the execution of successive primitive nodes that take fewer inputs from memory than store results in such memory. Since scheduling of primitive nodes is done by meshes, the largest memory requirement occurs when executing primitive nodes in a mesh that corresponds to an input boundary of the prism (i.e., execute nodes that store two results in memory but do not read two operands from memory). This property leads to the selection of meshes of size p by q, as indicated above.

With nodes scheduled as indicated in Figure 9b, storage required per cell is $pq + q = q(p+1)$. The outermost mesh of size p by q has q nodes that store two results in memory without reading any data from memory (the topmost horizontal path of the mesh), and $(p-1)q$ nodes that store two results but read only one operand from memory.

Storage access and organization. Each input to a primitive node is associated with flow of data along one of the axes of the graph. As indicated earlier, one of the three flows is implemented by a feedback loop within the cell. Let's assume that such flow is associated with axis X, as depicted in Figure 9b. The remaining two flows of data, along axes Y and Z, are obtained from memory, as shown also in Figure 9b. Consequently, local memory should be organized as two independent dual–ported modules that serve each of those two flows of data. Such modules have size pq and q respectively, each with bandwidth = 1 [word/time–step]. In such a case, memory modules are accessed without conflicts and with a simple pattern dictated by the scheduling of primitive nodes. These memories could also be implemented as queues.

Pipelined cells. Local memory facilitates the use of pipelining in cells. To accomplish such pipelining, it is necessary that there are independent operations that can be scheduled at successive time–steps. In a CMMDG, nodes that are interconnected by transmittent data correspond to independent nodes, because such dependency arises from broadcasting as shown in Figure 10. Scheduling primitive nodes by following such flow of transmittent data guarantees that independent operations exist, as long as the number of stages in the pipeline is less than or equal to the number of primitive nodes traversed by a transmittent data element within the G–node. Consequently, we choose a scheduling ordering as shown in Figure 10 and Figure 9b.

4.2 G–sets for a complete multi–mesh graph

We address now the division of the G–graph into G–sets. As stated earlier, G–sets are sets of as many G–nodes as there cells in an array. Moreover, G–nodes in a G–set should have the same computation time and should also have a dependency structure that matches the communication links in the array. Consequently, linear arrays require that G–sets be linear sets of G–nodes, while two–dimensional arrays require two–dimensional sets of G–nodes.

The G–graph for a CMMDG derived as indicated in the previous section is a regular graph, because all nodes have nearest–neighbor dependencies and the same computation time. Consequently, for an array with K cells, dividing such G–graph into G–sets is a simple process that consists of grouping K G–nodes in either linear or two–dimensional manner, as it was shown in Figure 7.

Dividing into G–sets determines the following characteristics of the array:

Number of G–sets $= n^2/(pqK)$, because there are $n^2/(pq)$ G–nodes that are divided into sets of size K.

Throughput of the array, given by $T^{-1} = n^2/(pqK)t_g$, where t_g is the computation time of G–nodes.

Utilization of the array, computed as $U = N/(KT^{-1})$, where N is the number of nodes in the fully–parallel dependency–graph and K is the number of G–nodes in a G–set (i.e., number of cells in the array).

4.3 Scheduling of G–sets in a complete multi–mesh graph

Scheduling of G–sets is performed as indicated in step 5 of our partitioning procedure.

4.4 Type of array as a function of partitioning

The type of array for an implementation is determined by the values of p and q in the partitioning method just described. The following table gives some examples:

Array	Systolic	Pseudo–systolic	Local–access (two–dimensional)
Partitioning method	Cut–and–pile	Coalescing and cut–and–pile	Coalescing
Values of p, q	$p = q = 1$	$p = q > 1$	$p = q = n/\sqrt{K}$
No. of G–sets	n^2/K	$n^2/(pqK)$	1
Storage per cell	2	$p(p+1)$	$n^2/K + n/\sqrt{K}$
Cell comm. bandwidth	1	$1/p$	$\sqrt{K}/n$

These expressions led to the values in Figure 2, where $p = q = \sqrt{S}$ in the case of pseudo–systolic arrays. The two words of storage per cell in a systolic array corresponds to registers required to latch input operands, so that actually there is no storage per cell. For $p = q = 1$, only cut–and–pile is used. On the other hand, for $p = q = n/\sqrt{K}$ there is one G–set so that only coalescing is used as the partitioning approach.

The expressions derived here allow us to develop a unified framework for partitioning by cut–and–pile, coalescing, and combination of the two approaches. Moreover, such framework permits the design of cells that trade memory and communication bandwidth, so that a designer can choose the partitioning approach that is more suitable for an implementation.

5 Partitioning triangularization by Givens' rotations

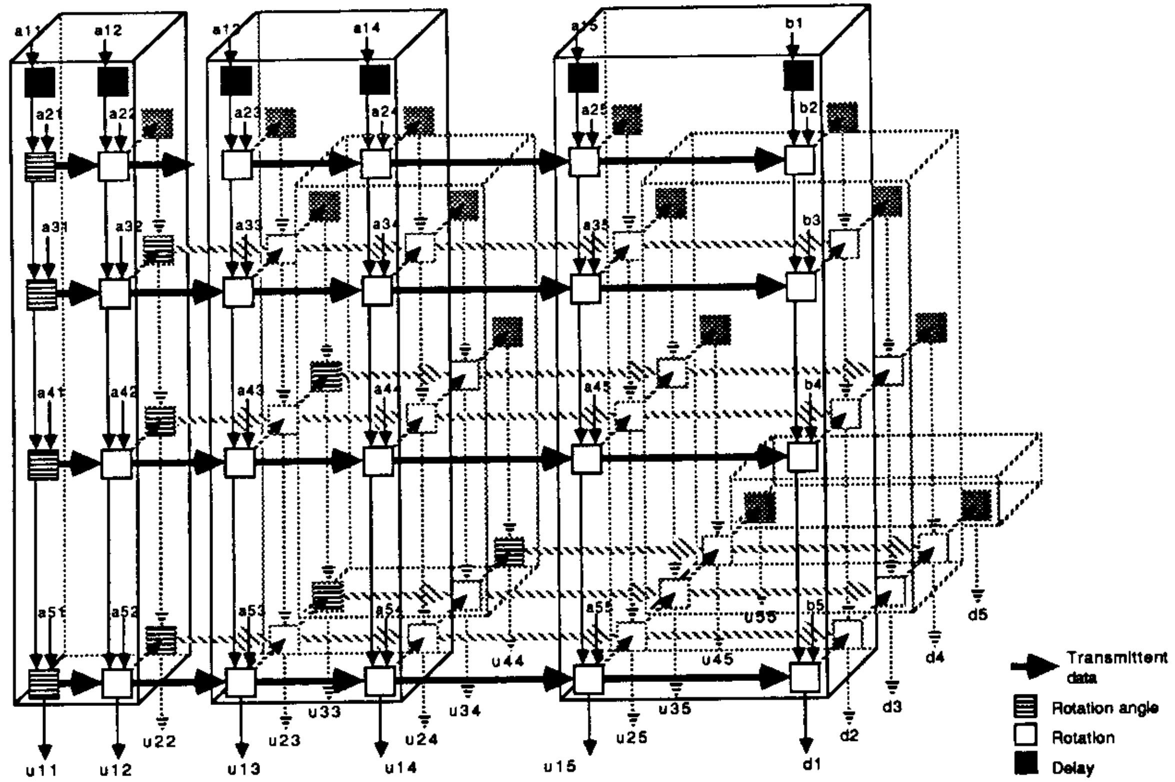
In the previous section, we have discussed partitioning a matrix algorithm that has a dependency structure corresponding to a complete multi–mesh dependency–graph, because such an algorithm has the most stringent requirements in terms of operations and communications for a given mesh size. Matrix multiplication is an example of such an algorithm. However, many important matrix computations have data dependency graphs that are not CMMDGs. Algorithms that aren't represented as CMMDGs include triangularization by Givens' rotations, LU–decomposition, transitive closure, Faddeev algorithm, Gaussian elimination, among others. In this section, we center our attention on such class of problems, using triangularization by Givens' rotations as an example. The multi–mesh graph for such computation was shown in Figure 5.

The application of our method to an algorithm that is not represented by a CMMDG is identical to the case of CMMDGs. Moreover, given the regularity of matrix algorithms, in many cases the internal portion of the multi–mesh dependency graph has the structure of a CMMDG. Consequently, multi–mesh graphs that are not CMMDGs are just particular cases of such CMMDGs.

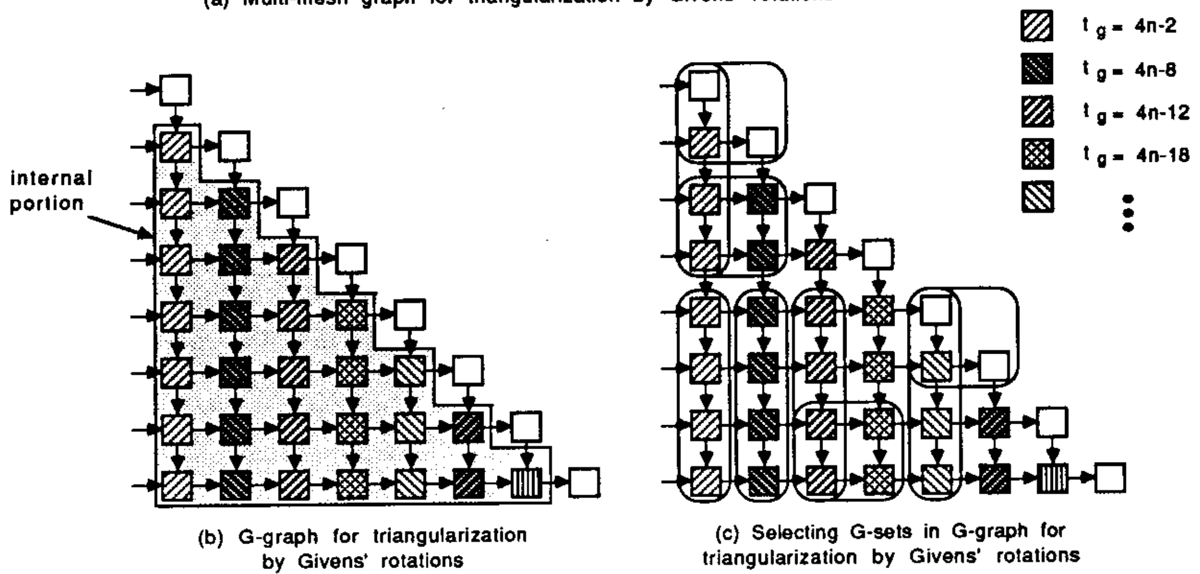
Deriving G–graph. As stated earlier, for good utilization of the resulting array, a G–graph should be derived in such a way that G–nodes have the same computation time. If this objective is not possible, one should try to obtain sets of neighbor G–nodes with the same computation time. These G–nodes later will constitute a G–set and will be executed concurrently in the array, so that good utilization is achieved. Since G–sets will be linear or two–dimensional sets of G–nodes, the derivation of the G–graph should try to obtain linear or two–dimensional sets of G–nodes with the same computation time, where the size of such sets is the same as the number of cells in the array.

The multi–mesh dependency–graph for triangularization by Givens' rotations, shown in Figure 5, has dependent meshes with decreasing number of primitive nodes. From the three possible alternatives to a derive a G–graph, drawing prisms as done in Figure 6 (i.e., along axis Z) leads to G–nodes with different computation time. On the other hand, drawing prisms along the other dimensions of the graph leads to linear sets of nodes with identical computation time. Figure 11a depicts one of the latter, where we have drawn prisms of size $p = q = 2$. The resulting G–graph is the triangular structure shown in Figure 11b, where the internal portion of the graph has been highlighted.[†] G–nodes in vertical paths within such internal portion have the same computation time.

[†]For space considerations, the multi–mesh graph shown in Figure 11a corresponds to a smaller problem than this G–graph.



(a) Multi-mesh graph for triangularization by Givens' rotations



(b) G-graph for triangularization by Givens' rotations

(c) Selecting G-sets in G-graph for triangularization by Givens' rotations

Figure 11: G–graph for triangularization by Givens' rotations

Dividing into G–sets. Dividing the G–graph derived above into G–sets is performed in the same way as in the case of the CMMDG. Such G–sets consist of either linear or two–dimensional sets of G–nodes, depending on the structure of the resulting array, as depicted in Figure 11c. Notice that two–dimensional G–sets include G–nodes with different computation time, so that utilization of the resulting two–dimensional array will not be maximal. On the other hand, given the properties of the G–graph, it is possible to select linear G–sets where all G–nodes have the same computation time, leading to better utilization of the corresponding linear array. Such linear G–sets correspond to vertical paths of the G–graph, as shown in Figure 11c.

Scheduling G–sets. Scheduling the G–sets derived above is performed the same as in the case of the CMMDG. Since input data appears only at leftmost G–nodes, and such data corresponds to communications with a host, we schedule G–sets in horizontal order. That is, we first schedule the leftmost G–set and then all G–sets to the right of that one. Upon reaching the rightmost end of the G–graph, we schedule the next G–set at the left of the graph and continue in the same manner.

The composition and scheduling of G–sets described above lead to linear and two–dimensional arrays that have the following properties:

- Depending on the size of the prisms (i.e., values of p and q), we obtain systolic or pseudo–systolic arrays with external memory, or local–access arrays. Pseudo–systolic arrays can use pipelined cells.
- For the same values of p and q, linear and two–dimensional arrays have the same cell communication bandwidth.
- Linear arrays have better utilization than two–dimensional ones, because they execute G–sets whose nodes have the same computation time.
- With the same number of cells K
  - Linear and two–dimensional arrays have the same I/O bandwidth to/from a host.
  - Computation time is the same in linear and two–dimensional arrays.
  - Throughput is higher in linear than two–dimensional arrays, because of the difference in utilization of cells.
  - Linear arrays require $K+1$ external memory modules while two–dimensional arrays require $2\sqrt{K}$ modules. However, the total memory capacity is the same in both cases.

From the properties above we can infer that the number of memory modules is the only advantage in cost or performance when partitioning an algorithm for execution in a two–dimensional array with respect to a linear one. On the other hand, linear arrays are simpler to implement and are better suited to incorporate fault–tolerant capabilities because it is easier to skip a faulty cell than to reconfigure a two–dimensional array. Consequently, we conclude that *a linear array offers better performance and implementation than two–dimensional array for partitioned execution of triangularization by Givens' rotations.* Such statement has also been shown true for other matrix algorithms [8,12].

The conclusions above are valid for large problems, that is when the size of a problem is much larger than the number of cells in an array. In such a case, it is possible to derive a two–dimensional

G–graph that will be "piled" onto an array in several G–sets. On the other hand, if the G–graph is not large enough then the amount of parallelism available in such G–graph (i.e., the number of G–nodes along a dimension of the graph) can't be exploited successfully with many cells.

Specifically, the conclusions stated above are valid if the G–graph has at least K by $\sqrt{K}$ G–nodes. In such a case, the G–graph is a two–dimensional graph that can be divided into linear or two–dimensional G–sets with K G–nodes, and tradeoffs between linear and two–dimensional arrays are possible. In other words, *the method is suitable to study tradeoffs between linear and two-dimensional arrays when applying limited coalescing and cut-and-pile combined, and not coalescing.* Such requirement implies that one of p or q must be less or equal than n/K and the other must be less or equal than $n/\sqrt{K}$.

6 Conclusions

We have addressed tradeoffs between local storage and cell communication bandwidth in the design of arrays for matrix computations. We have presented three types of arrays based on these two properties of cells. Such arrays include the systolic model of computation and two other models that we call pseudo–systolic and local–access.

We have presented a graph–based partitioning method to map matrix algorithms to the different types of arrays. This method is a transformational technique that uses a fully–parallel dependency–graph as the description of an algorithm. Such graph is transformed into another graph suitable for partitioning, which is later mapped onto an array. The method uses coalescing and cut–and–pile as partitioning approaches. With the method, it is possible to trade between local storage in a cell and cell communication bandwidth, thus reducing the communication bottleneck that characterizes systolic cells. Moreover, the method facilitates exploiting pipelining within cells. The resulting arrays exhibit high utilization, no overhead due to partitioning and simple control.

In addition, our method allows comparing linear and two–dimensional arrays for a given algorithm. We have shown that for a reasonably large problem, the advantages of a two–dimensional array over a linear array are limited to requiring fewer memory modules external to the array. In contrast, linear arrays might have better utilization and are more suitable to incorporate fault–tolerant features than two–dimensional structures.

With our method, a designer can determine the cell type required for an implementation based on the maximum values possible for cell communication bandwidth and functional unit computation rate, parameters that depend on the technology used. The ratio between these two values determines the amount of local memory needed in each cell. If such memory size is not feasible, then the designer must decide to reduce the computation rate or look for another technology that either allows larger local memory or higher cell communication bandwidth.

References

[1] J. Fortes, K. Fu, and B. Wah, "Systematic approaches to the design of algorithmically specified systolic arrays," in *Computer Architecture*, (V. Milutinović, ed.), pp. 454–494, North-Holland, 1988.

[2] H. Kung, "Why systolic architectures?," *IEEE Computer*, vol. 15, pp. 37–46, Jan. 1982.

[3] K. Bromley, S. Kung, and E. Swartzlander, eds., *International Conference on Systolic Arrays*, IEEE Computer Society Press, May 1988.

[4] M. Annaratone, E. Arnould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, , and J. Webb, "The Warp computer: architecture, implementation and performance," *IEEE Transactions on Computers*, vol. C-36, pp. 1523–1538, Dec. 1987.

[5] D. Foulser and R. Schreiber, "The Saxpy Matrix-1: a general purpose systolic computer," *IEEE Computer*, vol. 20, pp. 35–44, July 1987.

[6] H. Kung, "Systolic communication," in *International Conference on Systolic Arrays*, pp. 695–703, 1988.

[7] J. Navarro, J. Llaberia, and M. Valero, "Partitioning: an essential step in mapping algorithms into systolic array processors," *IEEE Computer*, vol. 20, pp. 77–89, July 1987.

[8] J. Moreno and T. Lang, "Graph–based partitioning of matrix algorithms for systolic arrays: application to transitive closure," in *International Conference on Parallel Processing*, 1988.

[9] S. Kung, *VLSI Array Processors*, p. 118. Prentice Hall, 1988.

[10] H. Ahmed, J. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *IEEE Computer*, vol. 15, pp. 65–82, Jan. 1982.

[11] J. Moreno and T. Lang, "Design of special–purpose arrays for matrix computations: preliminary results," in *SPIE Real–Time Signal Processing X*, pp. 53–65, 1987.

[12] J. Moreno and T. Lang, "On partitioning the Faddeev algorithm," in *International Conference on Systolic Arrays*, pp. 125–134, May 1988.