12. Systolic Systems

Systolic arrays probably constitute a perfect kind of special purpose computer. In their simplest appearance, they may provide only one operation, that is repeated over and over again. Yet, systolic arrays show an abundance of practice-oriented applications, mainly in fields dominated by iterative procedures: numerical mathematics, combinatorial optimisation, linear algebra, algorithmic graph theory, image and signal processing, speech and text processing, et cetera.

For a systolic array can be tailored to the structure of its one and only algorithm thus accurately! So that time and place of each executed operation are fixed once and for all. And communicating cells are permanently and directly connected, no switching required. The algorithm has in fact become hardwired. Systolic algorithms in this respect are considered to be ***hardware algorithms***.

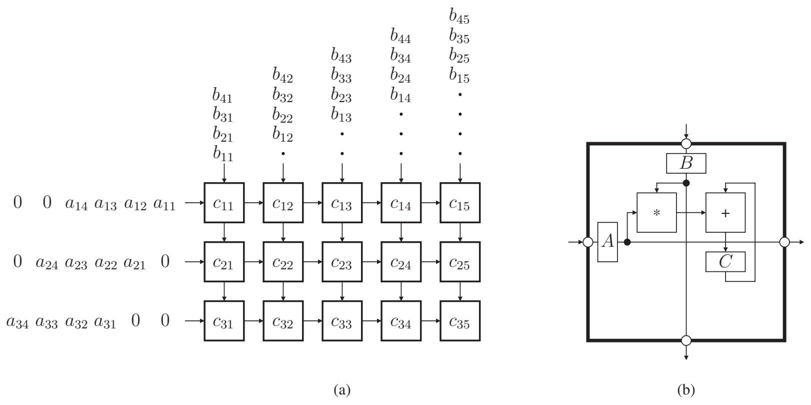
Please note that the term *systolic algorithms* usually does not refer to a set of concrete algorithms for solving a single specific computational problem, as for instance *sorting*. And this is quite in contrast to terms like *sorting algorithms*. Rather, systolic algorithms constitute a special style of specification, programming, and computation. So algorithms from many different areas of application can be *systolic* in style. But probably not all well-known algorithms from such an area might be suited to systolic computation.

Hence, this chapter does not intend to present *all* systolic algorithms, nor will it introduce even the most important systolic algorithms from any field of application. Instead, with a few simple but typical examples, we try to lay the foundations for the readers' general understanding of systolic algorithms.

The rest of this chapter is organised as follows: Section 12.1 shows some basic concepts of systolic systems by means of an introductory example. Section 12.2 explains how systolic arrays formally emerge from space-time transformations. Section 12.3 deals with input/output schemes. Section 12.4 is devoted to all aspects of control in systolic arrays. In section 12.5 we study the class of linear systolic arrays, raising further questions.

12.1. Basic concepts of systolic systems

The designation ***systolic*** follows from the operational principle of the systolic architecture. The systolic style is characterised by an intensive application of both *pipelining* and *pa-*



b_{41} b_{31} b_{21} b_{11}
b_{42} b_{32} b_{22} b_{12}
b_{43} b_{33} b_{23} b_{13}
b_{44} b_{34} b_{24} b_{14}
b_{45} b_{35} b_{25} b_{15}
0 0 a_{14} a_{13} a_{12} a_{11}
0 a_{24} a_{23} a_{22} a_{21} 0
a_{34} a_{33} a_{32} a_{31} 0 0
c_{11} c_{12} c_{13} c_{14} c_{15}
c_{21} c_{22} c_{23} c_{24} c_{25}
c_{31} c_{32} c_{33} c_{34} c_{35}
(a)
B
A
*
+
C
(b)

Figure 12.1. Rectangular systolic array for matrix product. **(a)** Array structure and input scheme. **(b)** Cell structure.

rallelism, controlled by a global and completely synchronous clock. ***Data streams*** pulsate rhythmically through the communication network, like streams of blood are driven from the heart through the veins of the body. Here, pipelining is not constrained to a single space axis but concerns *all data streams* possibly *moving in different directions* and *intersecting* in the cells of the systolic array.

A ***systolic system*** typically consists of a *host computer*, and the actual *systolic array*. Conceptionally, the host computer is of minor importance, just controlling the operation of the systolic array and supplying the data. The ***systolic array*** can be understood as a specialised network of cells rapidly performing data-intensive computations, supported by *massive parallelism*. A ***systolic algorithm*** is the program collaboratively executed by the cells of a systolic array.

Systolic arrays may appear very differently, but usually share a couple of key features: discrete time scheme, synchronous operation, regular (frequently two-dimensional) geometric layout, communication limited to directly neighbouring cells, and spartan control mechanisms.

In this section, we explain fundamental phenomena in context of systolic arrays, driven by a running example. A computational problem usually allows several solutions, each implemented by a specific systolic array. Among these, the most attractive designs (in whatever respect) may be very complex. Note, however, that in this educational text we are less interested in advanced solutions, but strive to present important concepts compactly and intuitively.

12.1.1. An introductory example: matrix product

Figure 12.1 shows a ***rectangular*** systolic array consisting of 15 ***cells*** for multiplying a $3\times N$ matrix A by an $N \times 5$ matrix B. The *parameter* N is not reflected in the *structure* of this particular systolic array, but in the *input scheme* and the *running time* of the algorithm.

The input scheme depicted is based on the special choice of parameter $N = 4$. Therefore, Figure 12.1 gives a solution to the following problem instance:

$$A \cdot B = C\ ,$$

where

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix},$$

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \end{pmatrix},$$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \end{pmatrix},$$

and

$$c_{ij} = \sum_{k=1}^{4} a_{ik} \cdot b_{kj} \qquad (1 \le i \le 3, 1 \le j \le 5)\ .$$

The cells of the systolic array can exchange data through ***links***, drawn as arrows between the cells in Figure 12.1(a). ***Boundary cells*** of the systolic array can also communicate with the *outside world*. All cells of the systolic array share a common ***connection pattern*** for communicating with their environment. The completely regular ***structure*** of the systolic array (placement and connection pattern of the cells) induces *regular data flows* along all connecting directions.

Figure 12.1(b) shows the ***internal structure*** of a cell. We find a *multiplier*, an *adder*, three *registers*, and four *ports*, plus some wiring between these units. Each ***port*** represents an interface to some external link that is attached to the cell. All our cells are of the same structure.

Each of the registers A, B, C can store a single data item. The designations of the registers are suggestive here, but arbitrary in principle. Registers A and B get their values from ***input ports***, shown in Figure 12.1(b) as small circles on the left resp. upper border of the cell.

The current values of registers A and B are used as operands of the multiplier and, at the same time, are passed through ***output ports*** of the cell, see the circles on the right resp. lower border. The result of the multiplication is supplied to the adder, with the second operand originating from register C. The result of the addition eventually overwrites the past value of register C.

12.1.2. Problem parameters and array parameters

The 15 cells of the systolic array are organised as a rectangular pattern of three rows by five columns, exactly as with matrix C. Also, these dimensions directly correspond to the number of rows of matrix A and the number of columns of matrix B. The ***size of the systolic***

array, therefore, *corresponds* to the *size of some data structures* for the problem to solve. If we had to multiply an $N_1 \times N_3$ matrix A by an $N_3 \times N_2$ matrix B in the general case, then we would need a systolic array with N_1 rows and N_2 columns.

The quantities N_1, N_2, N_3 are parameters of the problem to solve, because the number of operations to perform depends on each of them; they are thus ***problem parameters***. The size of the systolic array, in contrast, depends on the quantities N_1 and N_2, only. For this reason, N_1 and N_2 become also ***array parameters***, for this particular systolic array, whereas N_3 is *not* an array parameter.

Remark. For matrix product, we will see another systolic array in section 12.2, with dimensions dependent on all three problem parameters N_1, N_2, N_3.

An $N_1 \times N_2$ systolic array as shown in Figure 12.1 would also permit to multiply an $M_1 \times M_3$ matrix A by an $M_3 \times M_2$ matrix B, where $M_1 \leq N_1$ and $M_2 \leq N_2$. This is important if we intend to use the same systolic array for the multiplication of matrices of varying dimensions. Then we would operate on a properly dimensioned rectangular subarray, only, consisting of M_1 rows and M_2 columns, and located, for instance, in the upper left corner of the complete array. The remaining cells would also work, but without any contribution to the solution of the whole problem; they should do no harm, of course.

12.1.3. Space coordinates

Now let's assume that we want to assign unique ***space coordinates*** to each cell of a systolic array, for characterising the geometric position of the cell relative to the whole array. In a *rectangular* systolic array, we simply can use the respective row and column numbers, for instance. The cell marked with c_{11} in Figure 12.1 thus would get the coordinates $(1, 1)$, the cell marked with c_{12} would get the coordinates $(1, 2)$, cell c_{21} would get $(2, 1)$, and so on. For the remainder of this section, we take space coordinates constructed in such a way for granted.

In principle it does not matter where the coordinate origin lies, where the axes are pointing to, which direction in space corresponds to the first coordinate, and which to the second. In the system presented above, the order of the coordinates has been chosen corresponding to the designation of the matrix components. Thus, the first coordinate stands for the rows numbered top to bottom from position 1, the second component stands for the columns numbered left to right, also from position 1.

Of course, we could have made a completely different choice for the coordinate system. But the presented system perfectly matches our particular systolic array: the indices of a matrix element c_{ij} computed in a cell agree with the coordinates of this cell. The entered rows of the matrix A carry the same number as the first coordinate of the cells they pass; correspondingly for the second coordinate, concerning the columns of the matrix B. All links (and thus all passing data flows) are in parallel to some axis, and towards ascending coordinates.

It is not always so clear how expressive space coordinates can be determined; we refer to the systolic array from Figure 12.3(a) as an example. But whatsoever the coordinate system is chosen: it is important that the regular structure of the systolic array is obviously reflected in the coordinates of the cells. Therefore, almost always integral coordinates are used. Moreover, the coordinates of cells with minimum Euclidean distance should differ in one component, only, and then with distance 1.

12.1.4. Serialising generic operators

Each active cell (i, j) from Figure 12.1 computes exactly the element c_{ij} of the result matrix C. Therefore, the cell must evaluate the ***dot product***

$$\sum_{k=1}^{4} a_{ik} \cdot b_{kj} \ .$$

This is done iteratively: in each step, a product $a_{ik} \cdot b_{kj}$ is calculated and added to the current partial sum for c_{ij}. Obviously, the partial sum has to be *cleared*—or set to another initial value, if required—before starting the accumulation. Inspired by the classical notation of imperative programming languages, the general proceeding could be specified in pseudocode as follows:

```
Matrix-Product($N_1, N_2, N_3$)

1  for $i \leftarrow 1$ to $N_1$
2     do for $j \leftarrow 1$ to $N_2$
3           do $c(i, j) \leftarrow 0$
4              for $k \leftarrow 1$ to $N_3$
5                 do $c(i, j) \leftarrow c(i, j) + a(i, k) \cdot b(k, j)$
6  return $C$
```

If $N_1 = N_2 = N_3 = N$, we have to perform N^3 multiplications, additions, and assignments, each. Hence the *running time* of this algorithm is of order $\Theta(N^3)$ for any sequential processor.

The sum operator $\sum$ is one of the so-called ***generic operators***, that combine an arbitrary number of operands. In the systolic array from Figure 12.1, all additions contributing to a particular sum are performed in the same cell. However, there are plenty of examples where the individual operations of a generic operator are spread over several cells—see, for instance, the systolic array from Figure 12.3.

Remark. Further examples of generic operators are: product, minimum, maximum, as well as the Boolean operators AND, OR, and EXCLUSIVE OR.

Thus, generic operators usually have to be ***serialised*** before the calculations to perform can be assigned to the cells of the systolic array. Since the distribution of the individual operations to the cells is not unique, generic operators generally must be dealt with in another way than simple operators with fixed arity, as for instance the dyadic addition.

12.1.5. Assignment-free notation

Instead of using an imperative style as in algorithm Matrix-product, we better describe systolic programs by an ***assignment-free notation*** which is based on an *equational calculus*. Thus we avoid *side effects* and are able to *directly express parallelism*. For instance, we may be bothered about the reuse of the program variable $c(i, j)$ from algorithm Matrix-product. So, we replace $c(i, j)$ with a sequence of ***instances*** $c(i, j, k)$, that stand for the successive states of $c(i, j)$. This approach yields a so-called ***recurrence equation***. We are now able to state the general matrix product from algorithm Matrix-product by the following assignment-free expressions:

$$
\begin{array}{ll}
\textit{input operations} & \\
c(i,j,0) = 0 & 1 \le i \le N_1, 1 \le j \le N_2. \\
& \\
\textit{calculations} & \\
c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3. \\
& \\
\textit{output operations} & \\
c_{ij} = c(i,j,N_3) & 1 \le i \le N_1, 1 \le j \le N_2 .
\end{array}
\tag{12.1}
$$

System (12.1) explicitly describes the fine structure of the executed systolic algorithm. The first equation specifies all ***input data***, the third equation all ***output data***. The systolic array implements these equations by ***input/output operations***. Only the second equation corresponds to real calculations.

Each equation of the system is accompanied, on the right side, by a ***quantification***. The quantification states the set of values the ***iteration variables*** i and j (and, for the second equation, also k) should take. Such a set is called a ***domain***. The iteration variables i, j, k of the second equation can be combined in an ***iteration vector*** (i, j, k). For the input/output equations, the iteration vector would consist of the components i and j, only. To get a closed representation, we augment this vector by a third component k, that takes a fixed value. Inputs then are characterised by $k = 0$, outputs by $k = N_3$. Overall we get the following system:

$$
\begin{array}{ll}
\textit{input operations} & \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0. \\
& \\
\textit{calculations} & \\
c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3. \\
& \\
\textit{output operations} & \\
c_{ij} = c(i,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3 .
\end{array}
\tag{12.2}
$$

Note that although the domains for the input/output equations now are formally also of dimension 3, as a matter of fact they are only two-dimensional in the classical geometric sense.

12.1.6. Elementary operations

From equations as in system (12.2), we directly can infer the atomic entities to perform in the cells of the systolic array. We find these operations by instantiating each equation of the system with all points of the respective domain. If an equation contains several suboperations corresponding to one point of the domain, these are seen as a ***compound operation***, and are always processed together by the same cell in one working cycle.

In the second equation of system (12.2), for instance, we find the multiplication

$a(i,k) \cdot b(k,j)$ and the successive addition $c(i,j,k) = c(i,j,k-1) + \cdots$. The corresponding ***elementary operations***—multiplication and addition—are indeed executed together as a *multiply-add* compound operation by the cell of the systolic array shown in Figure 12.1(b).

Now we can assign a designation to each elementary operation, also called *coordinates*. A straight-forward method to define suitable coordinates is provided by the iteration vectors (i, j, k) used in the quantifications.

Applying this concept to system (12.1), we can for instance assign the tuple of coordinates (i, j, k) to the calculation $c(i, j, k) = c(i, j, k-1) + a(i,k) \cdot b(k, j)$. The same tuple (i, j, k) is assigned to the input operation $c(i, j, k) = 0$, but with setting $k = 0$. By the way: all domains are disjoint in this example.

If we always use the iteration vectors as designations for the calculations and the input/output operations, there is no further need to distinguish between coordinates and iteration vectors. Note, however, that this decision also mandates that all operations belonging to a certain point of the domain together constitute a compound operation—even when they appear in different equations and possibly are not related. For simplicity, we always use the iteration vectors as coordinates in the sequel.

12.1.7. Discrete timesteps

The various elementary operations always happen in ***discrete timesteps*** in the systolic cells. All these timesteps driving a systolic array are of equal duration. Moreover, all cells of a systolic array work completely ***synchronous***, i.e., they all start and finish their respective communication and calculation steps at the same time. Successive timesteps controlling a cell seamlessly follow each other.

Remark. But haven't we learned from Albert Einstein that strict simultaneity is physically impossible? Indeed, all we need here are cells that operate almost simultaneously. Technically this is guaranteed by providing to all systolic cells a common ***clock signal*** that switches all registers of the array. Within the bounds of the usually achievable accuracy, the communication between the cells happens sufficiently synchronised, and thus no loss of data occurs concerning send and receive operations. Therefore, it should be justified to assume a conceptional simultaneity for theoretical reasoning.

Now we can slice the physical time into units of a timestep, and number the timesteps consecutively. The origin on the time axis can be arbitrarily chosen, since time is running synchronously for all cells. A reasonable decision would be to take $t = 0$ as the time of the first input in any cell. Under this regime, the elementary compound operation of system (12.1) designated by (i, j, k) would be executed at time $i + j + k - 3$. On the other hand, it would be evenly justified to assign the time $i + j + k$ to the coordinates (i, j, k); because this change would only induce a global time shift by three time units.

So let us assume for the following that the execution of an instance (i, j, k) starts at time $i + j + k$. The first calculation in our example then happens at time $t = 3$, the last at time $t = N_1 + N_2 + N_3$. The *running time* thus amounts to $N_1 + N_2 + N_3 - 2$ timesteps.

12.1.8. External and internal communication

Normally, the data needed for calculation by the systolic array initially are not yet located inside the cells of the array. Rather, they must be infused into the array from the ***outside***

world. The outside world in this case is a ***host computer***, usually a *scalar control processor* accessing a central *data storage*. The control processor, at the right time, fetches the necessary data from the storage, passes them to the systolic array in a suitable way, and eventually writes back the calculated results into the storage.

Each cell (i, j) must access the operands a_{ik} and b_{kj} during the timestep concerning index value k. But only the cells of the leftmost column of the systolic array from Figure 12.1 get the items of the matrix A directly as *input data* from the outside world. All other cells must be provided with the required values a_{ik} from a neighbouring cell. This is done via the horizontal *links* between neighbouring cells, see Figure 12.1(a). The item a_{ik} successively passes the cells $(i, 1)$, $(i, 2)$, …, (i, N_2). Correspondingly, the value b_{kj} enters the array at cell $(1, j)$, and then flows through the vertical links, reaching the cells $(2, j)$, $(3, j)$, … up to cell (N_1, j). An arrowhead in the Figure shows in which *direction* the link is oriented.

Frequently, it is considered problematic to transmit a value over large distances within a single *timestep*, in a distributed or parallel architecture. Now suppose that, in our example, cell (i, j) got the value a_{ik} during timestep t from cell $(i, j-1)$, or from the outside world. For the reasons described above, a_{ik} is not passed from cell (i, j) to cell $(i, j+1)$ in the same timestep t, but one timestep later, i.e., at time $t+1$. This also holds for the values b_{kj}. The ***delay*** is visualised in the detail drawing of the cell from Figure 12.1(b): input data flowing through a cell always pass one register, and each passed register induces a delay of exactly one timestep.

Remark. For systolic architectures, it is mandatory that any path between two cells contains at least one register—even when forwarding data to a neighbouring cell, only. All registers in the cells are synchronously switched by the global clock signal of the systolic array. This results in the characteristic rhythmical traffic on all links of the systolic array. Because of the analogy with pulsating veins, the medical term *systole* has been reused for the name of the concept.

To elucidate the delayed forwarding of values, we augment system (12.1) with further equations. Repeatedly *used* values like a_{ik} are represented by separate *instances*, one for each access. The result of this proceeding—that is very characteristic for the design of systolic algorithms—is shown as system (12.3).

$$
\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k) = a_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3\ , \\
b(i,j,k) = b_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3\ , \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0\ . \\
 & \\
\textit{calculations and forwarding} & \\
a(i,j,k) = a(i,j-1,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ , \\
b(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ , \\
c(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ . \\
\quad + a(i,j-1,k) \cdot b(i-1,j,k) & \\
 & \\
\textit{output operations} & \\
c_{ij} = c(i,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3\ .
\end{array}
\tag{12.3}
$$

Each of the partial sums $c(i, j, k)$ in the progressive evaluation of c_{ij} is calculated in a certain timestep, and then used only once, namely in the next timestep. Therefore, cell (i, j) must provide a register (named C in Figure 12.1(b)) where the value of $c(i, j, k)$ can be stored for one timestep. Once the old value is no longer needed, the register holding $c(i, j, k)$ can be overwritten with the new value $c(i, j, k + 1)$. When eventually the dot product is completed, the register contains the value $c(i, j, N_3)$, that is the final result c_{ij}. Before performing any computation, the register has to be ***cleared***, i.e., preloaded with a zero value—or any other desired value.

In contrast, there is no need to store the values a_{ik} and b_{kj} permanently in cell (i, j). As we can learn from Figure 12.1(a), each row of the matrix A is delayed by one timestep with respect to the preceding row. And so are the columns of the matrix B. Thus the values $a(i, j - 1, k)$ and $b(i - 1, j, k)$ arrive at cell (i, j) exactly when the calculation of $c(i, j, k)$ is due. They are put to the registers A resp. B, then immediately fetched from there for the multiplication, and in the same cycle forwarded to the neighbouring cells. The values a_{ik} and b_{kj} are of no further use for cell (i, j) after they have been multiplied, and need not be stored there any longer. So A and B are overwritten with new values during the next timestep.

It should be obvious from this exposition that we urgently need to make economic use of the memory contained in a cell. Any calculation and any communication must be coordinated in space and time in such a way that storing of values is limited to the shortest-possible time interval. This goal can be achieved by immediately using and forwarding the received values. Besides the overall structure of the systolic array, choosing an appropriate *input/output scheme* and placing the corresponding number of *delays* in the cells essentially facilitates the desired coordination. Figure 12.1(b) in this respect shows the smallest possible delay by one timestep.

Geometrically, the input input scheme of the example resulted from *skewing* the matrices A and B. Thereby some places in the ***input streams*** for matrix A became vacant and had to be filled with zero values; otherwise, the calculation of the c_{ij} would have been garbled. The input streams in length depend on the *problem parameter* N_3.

As can been seen in Figure 12.1, the items of matrix C are calculated ***stationary***, i.e., all additions contributing to an item c_{ij} happen in the same cell. ***Stationary variables*** don't move at all during the calculation in the systolic array. Stationary results eventually must be forwarded to a ***border*** of the array in a supplementary action for getting delivered to the outside world. Moreover, it is necessary to initialise the register for item c_{ij}. Performing these extra tasks requires a high expenditure of runtime and hardware. We will further study this problem in section 12.4.

12.1.9. Pipelining

The characteristic operating style with globally synchronised discrete timesteps of equal duration and the strict separation in time of the cells by registers suggest systolic arrays to be special cases of *pipelined* systems. Here, the registers of the cells correspond to the well-known *pipeline registers*. However, classical pipelines come as linear structures, only, whereas systolic arrays frequently extend into more spatial dimensions—as visible in our example. A *multi-dimensional systolic array* can be regarded as a set of interconnected linear pipelines, with some justification. Hence it should be apparent that basic properties of one-dimensional pipelining also apply to multi-dimensional systolic arrays.

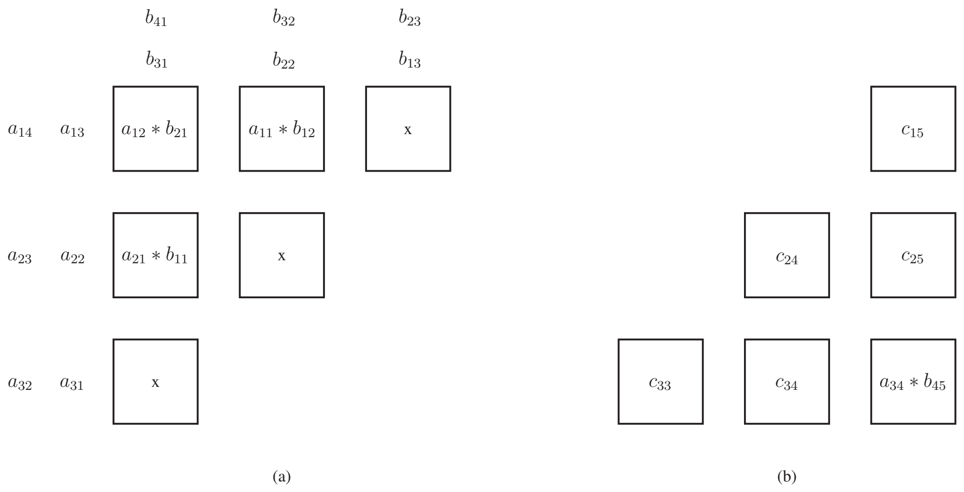


Figure 12.2. Two *snapshots* for the systolic array from Figure 12.1.

A typical effect of pipelining is the reduced ***utilisation*** at startup and during shut-down of the operation. Initially, the pipe is empty, no pipeline stage active. Then, the first stage receives data and starts working; all other stages are still idle. During the next timestep, the first stage passes data to the second stage and itself receives new data; only these two stages do some work. More and more stages become active until all stages process data in every timestep; the pipeline is now fully utilised for the first time. After a series of timesteps at maximum load, with duration dependent on the length of the data stream, the input sequence ceases; the first stage of the pipeline therefore runs out of work. In the next timestep, the second stage stops working, too. And so on, until eventually all stages have been fallen asleep again. Phases of reduced activity diminish the average performance of the whole pipeline, and the relative contribution of this drop in productivity is all the worse, the more stages the pipeline has in relation to the length of the data stream.

We now study this phenomenon to some depth by analysing the two-dimensional systolic array from Figure 12.1. As expected, we find a lot of idling cells when starting or finishing the calculation. In the first timestep, only cell $(1, 1)$ performs some useful work; all other cells in fact do calculations that work like null operations—and that's what they are supposed to do in this phase. In the second timestep, cells $(1, 2)$ and $(2, 1)$ come to real work, see Figure 12.2(a). Data is flooding the array until eventually all cells are doing work. After the last true data item has left cell $(1, 1)$, the latter is no longer contributing to the calculation but merely reproduces the finished value of c_{11}. Step by step, more and more cells drop off. Finally, only cell (N_1, N_2) makes a last necessary computation step; Figure 12.2(b) shows this concluding timestep.

Exercises

12.1-1 What must be changed in the input scheme from Figure 12.1(a) to multiply a 2×6 matrix by a 6×3 matrix on the same systolic array? Could the calculations be organised

such that the result matrix would emerge in the lower right corner of the systolic array?
12.1-2 Why is it necessary to clear spare slots in the input streams for matrix A, as shown in Figure 12.1? Why haven't we done the same for matrix B also?
12.1-3 If the systolic array from Figure 12.1 should be interpreted as a pipeline: how many stages would you suggest to adequately describe the behaviour?

12.2. Space-time transformation and systolic arrays

Although the approach taken in the preceding section should be sufficient for a basic understanding of the topic, we have to work harder to describe and judge the properties of systolic arrays in a quantitative and precise way. In particular the solution of *parametric problems* requires a solid mathematical framework. So, in this section, we study central concepts of a formal theory on *uniform algorithms*, based on *linear transformations*.

12.2.1. Further example: matrix product without stationary variables

System (12.3) can be computed by a multitude of other systolic arrays, besides that from Figure 12.1. In Figure 12.3, for example, we see such an alternative systolic array. Whereas the same function is evaluated by both architectures, the appearance of the array from Figure 12.3 is very different:

- The number of cells now is considerably larger, altogether 36, instead of 15.
- The shape of the array is ***hexagonal***, instead of rectangular.
- Each cell now has three input ports and three output ports.
- The input scheme is clearly different from that of Figure 12.1(a).
- And finally: the matrix C here also flows through the whole array.

The cell structure from Figure 12.3(b) at first view does not appear essentially distinguished from that in Figure 12.1(b). But the differences matter: there are no *cyclic paths* in the new cell, thus *stationary variables* can no longer appear. Instead, the cell is provided with three input ports and three output ports, passing items of all three matrices through the cell. The direction of communication at the ports on the right and left borders of the cell has changed, as well as the assignment of the matrices to the ports.

12.2.2. The space-time transformation as a global view

How system (12.3) is related to Figure 12.3? No doubt that you were able to fully understand the operation of the systolic array from Section 12.1 without any special aid. But for the present example this is considerably more difficult—so now you may be sufficiently motivated for the use of a mathematical formalism.

We can assign two fundamental measures to each elementary operation of an algorithm for describing the execution in the systolic array: the time *when* the operation is performed, and the position of the cell *where* the operation is performed. As will become clear in the sequel, after fixing the so-called *space-time transformation* there are hardly any degrees of freedom left for further design: practically all features of the intended systolic array strictly follow from the chosen space-time transformation.

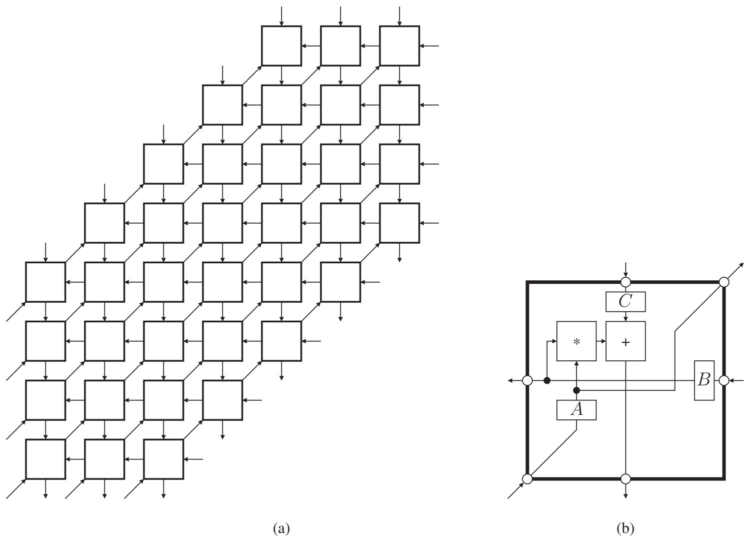


Figure 12.3. Hexagonal systolic array for matrix product. **(a)** Array structure and principle of the data input/output. **(b)** Cell structure.

As for the systolic array from Figure 12.1, the execution of an instance (i, j, k) in the systolic array from Figure 12.3 happens at time $t = i + j + k$. We can represent this expression as the dot product of a ***time vector***

$$\pi = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \tag{12.4}$$

by the iteration vector

$$v = \begin{pmatrix} i & j & k \end{pmatrix} , \tag{12.5}$$

hence

$$t = \pi \cdot v ; \tag{12.6}$$

so in this case

$$t = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = i + j + k . \tag{12.7}$$

The space coordinates $z = (x, y)$ of the executed operations in the example from Figure 12.1 can be inferred as $z = (i, j)$ from the iteration vector $v = (i, j, k)$ according to our decision in Section 12.1.3. The chosen map is a ***projection*** of the space $\mathbb{R}^3$ along the k axis. This linear map can be described by a ***projection matrix***

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} . \tag{12.8}$$

To find the space coordinates, we multiply the projection matrix P by the iteration vector

v, written as

$$z = P \cdot v \; . \tag{12.9}$$

The ***projection direction*** can be represented by any vector u perpendicular to all rows of the projection matrix,

$$P \cdot u = \vec{0} \; . \tag{12.10}$$

For the projection matrix P from (12.8), one of the possible ***projection vectors*** would be $u = (0, 0, 1)$.

Projections are very popular for describing the space coordinates when designing a systolic array. Also in our example from Figure 12.3(a), the space coordinates are generated by projecting the iteration vector. Here, a feasible projection matrix is given by

$$P = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \; . \tag{12.11}$$

A corresponding projection vector would be $u = (1, 1, 1)$.

We can combine the projection matrix and the time vector in a matrix T, that fully describes the ***space-time transformation***,

$$\begin{pmatrix} z \\ t \end{pmatrix} = \begin{pmatrix} P \\ \pi \end{pmatrix} \cdot v = T \cdot v \; . \tag{12.12}$$

The first and second rows of T are constituted by the projection matrix P, the third row by the time vector π.

For the example from Figure 12.1, the matrix T giving the space-time transformation reads as

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \; ; \tag{12.13}$$

for the example from Figure 12.3 we have

$$T = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \; . \tag{12.14}$$

Space-time transformations may be understood as a *global view* to the systolic system. Applying a space-time transformation—that is linear, here, and described by a matrix T—to a system of recurrence equations directly yields the external features of the systolic array, i.e., its ***architecture***—consisting of space coordinates, connection pattern, and cell structure.

Remark. Instead of purely linear maps, we alternatively may consider general affine maps, additionally providing a translative component, $T \cdot v + h$. Though as long as we treat all iteration vectors with a common space-time transformation, affine maps are not really required.

12.2.3. Parametric space coordinates

If the domains are numerically given and contain few points in particular, we can easily calculate the concrete set of space coordinates via equation (12.9). But when the domains are specified parametrically as in system (12.3), the positions of the cells must be determined

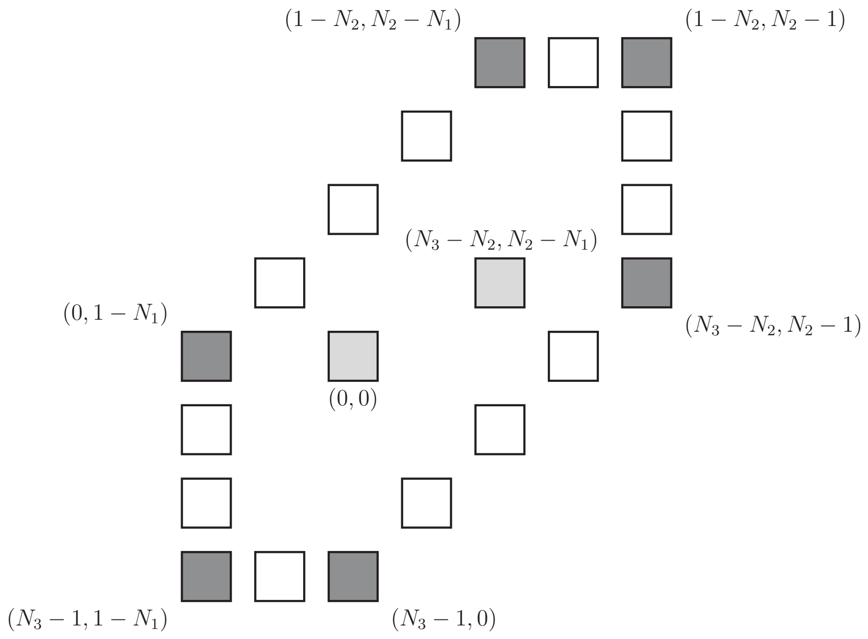


Figure 12.4. Image of a rectangular domain under projection. Most interior points have been suppressed for clarity. Images of previous vertex points are shaded.

by *symbolic evaluation*. The following explanation especially dwells on this problem.

Suppose that each cell of the systolic array is represented geometrically by a point with space coordinates $z = (x, y)$ in the two-dimensional space $\mathbb{R}^2$. From each iteration vector v of the domain S, by equation (12.9) we get the space coordinates z of a certain processor, $z = P \cdot v$: the operations denoted by v are projected onto cell z. The set $P(S) = \{P \cdot v : v \in S\}$ of space coordinates states the positions of all cells in the systolic array necessary for correct operation.

To our advantage, we normally use domains that can be described as the set of all integer points inside a convex region, here a subset of $\mathbb{R}^3$—called ***dense convex domains***. The convex hull of such a domain with a finite number of domain points is a *polytope*, with domain points as vertices. Polytopes map to polytopes again by arbitrary linear transformations. Now we can make use of the fact that each projection is a linear transformation. Vertices of the destination polytope then are images of vertices of the source polytope.

Remark. But not all vertices of a source polytope need to be projected to vertices of the destination polytope, see for instance Figure 12.4.

When projected by an integer matrix P, the *lattice* $\mathbf{Z}^3$ maps to the lattice $\mathbf{Z}^2$ if P can be extended by an integer time vector π to a *unimodular* space-time matrix T. Practically any dense convex domain, apart from some exceptions irrelevant to usual applications, thereby maps to another dense convex set of space coordinates, that is completely characterised by the vertices of the hull polytope. To determine the *shape* and the *size* of the systolic array, it is therefore sufficient to apply the matrix P to the vertices of the convex hull of S.

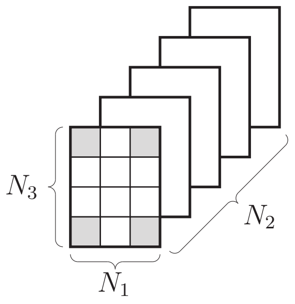


Figure 12.5. Partitioning of the space coordinates.

Remark. Any square integer matrix with determinant ±1 is called ***unimodular***. Unimodular matrices have unimodular inverses.

We apply this method to the integer domain

$$S = [1, N_1] \times [1, N_2] \times [1, N_3] \tag{12.15}$$

from system (12.3). The vertices of the convex hull here are

$$\begin{gathered}(1,1,1),(N_1,1,1),(1,N_2,1),(1,1,N_3),\\(1,N_2,N_3),(N_1,1,N_3),(N_1,N_2,1),(N_1,N_2,N_3)\ .\end{gathered} \tag{12.16}$$

For the projection matrix P from (12.11), the vertices of the corresponding image have the positions

$$\begin{gathered}(N_3-1,0),(N_3-1,1-N_1),(0,1-N_1),\\(1-N_2,N_2-N_1),(1-N_2,N_2-1),(N_3-N_2,N_2-N_1)\ .\end{gathered} \tag{12.17}$$

Since S has eight vertices, but the image $P(S)$ only six, it is obvious that two vertices of S have become *interior points* of the image, and thus are of no relevance for the size of the array; namely the vertices $(1,1,1)$ and (N_1, N_2, N_3). This phenomenon is sketched in Figure 12.4.

The settings $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$ yield the vertices $(3,0)$, $(3,-2)$, $(0,-2)$, $(-4,2)$, $(-4,4)$, and $(-1,4)$. We see that space coordinates in principle can be negative. Moreover, the choice of an origin—that here lies in the interior of the polytope—might not always be obvious.

As the image of the projection, we get a systolic array with *hexagonal* shape and parallel opposite borders. On these, we find N_1, N_2, and N_3 integer points, respectively; cf. Figure 12.5. Thus, as opposed to our first example, *all* problem parameters here are also array parameters.

The area function of this region is of order $\Theta(N_1 \cdot N_2 + N_1 \cdot N_3 + N_2 \cdot N_3)$, and thus depends on all three matrix dimensions. So this is quite different from the situation in Figure 12.1(a), where the area function—for the same problem—is of order $\Theta(N_1 \cdot N_2)$.

Improving on this approximate calculation, we finally count the exact number of cells. For this process, it might be helpful to partition the entire region into subregions for which the number of cells comprised can be easily determined; see Figure 12.5. The points $(0, 0)$, $(N_3 - 1, 0)$, $(N_3 - 1, 1 - N_1)$, and $(0, 1 - N_1)$ are the vertices of a rectangle with $N_1 \cdot N_3$ cells. If we translate this point set up by $N_2 - 1$ cells and right by $N_2 - 1$ cells, we exactly cover the whole region. Each shift by one cell up and right contributes just another $N_1 + N_3 - 1$ cells. Altogether this yields $N_1 \cdot N_3 + (N_2 - 1) \cdot (N_1 + N_3 - 1) = N_1 \cdot N_2 + N_1 \cdot N_3 + N_2 \cdot N_3 - (N_1 + N_2 + N_3) + 1$ cells.

For $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$ we thereby get a number of 36 cells, as we have already learned from Figure 12.3(a).

12.2.4. Symbolically deriving the running time

The running time of a systolic algorithm can be symbolically calculated by an approach similar to that in section 12.2.3. The time transformation according to formula (12.6) as well is a linear map. We find the timesteps of the first and the last calculations as the minimum resp. maximum in the set $\pi(S) = \{\pi \cdot v : v \in S\}$ of execution timesteps. Following the discussion above, it thereby suffices to vary v over the vertices of the convex hull of S.

The ***running time*** is then given by the formula

$$t_\Sigma = 1 + \max P(S) - \min P(S) \ . \tag{12.18}$$

Adding one is mandatory here, since the first as well as the last timestep belong to the calculation.

For the example from Figure 12.3, the vertices of the polytope as enumerated in (12.16) are mapped by (12.7) to the set of images

$$\{3, 2 + N_1, 2 + N_2, 2 + N_3, 1 + N_1 + N_2, 1 + N_1 + N_3, 1 + N_2 + N_3, N_1 + N_2 + N_3\}qkoz.$$

With the basic assumption $N_1, N_2, N_3 \geq 1$, we get a minimum of 3 and a maximum of $N_1 + N_2 + N_3$, thus a running time of $N_1 + N_2 + N_3 - 2$ timesteps, as for the systolic array from Figure 12.1—no surprise, since the domains and the time vectors agree.

For the special problem parameters $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$, a running time of $12 - 3 + 1 = 10$ timesteps can be derived.

If $N_1 = N_2 = N_3 = N$, the systolic algorithm shows a running time of order $\Theta(N)$, using $\Theta(N^2)$ systolic cells.

12.2.5. How to unravel the communication topology

The ***communication topology*** of the systolic array is induced by applying the space-time transformation to the *data dependences* of the algorithm. Each data dependence results from a direct use of a variable instance to calculate another instance of the same variable, or an instance of another variable.

Remark. In contrast to the general situation where a data dependence analysis for imperative programming languages has to be performed by highly optimising compilers, data dependences here always are *flow dependences*. This is a direct consequence from the assignment-free notation employed by us.

The ***data dependences*** can be read off the quantified equations in our assignment-free notation by comparing their right and left sides. For example, we first analyse the equation

$c(i, j, k) = c(i, j, k-1) + a(i, j-1, k) \cdot b(i-1, j, k)$ from system (12.3).

The value $c(i, j, k)$ is calculated from the values $c(i, j, k-1)$, $a(i, j-1, k)$, and $b(i-1, j, k)$. Thus we have a ***data flow*** from $c(i, j, k-1)$ to $c(i, j, k)$, a data flow from $a(i, j-1, k)$ to $c(i, j, k)$, and a data flow from $b(i-1, j, k)$ to $c(i, j, k)$.

All properties of such a data flow that matter here can be covered by a ***dependence vector***, which is the iteration vector of the calculated variable instance minus the iteration vector of the correspondingly used variable instance.

The iteration vector for $c(i, j, k)$ is (i, j, k); that for $c(i, j, k-1)$ is $(i, j, k-1)$. Thus, as the difference vector, we find

$$d_C = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i \\ j \\ k-1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} . \qquad (12.19)$$

Correspondingly, we get

$$d_A = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i \\ j-1 \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad (12.20)$$

and

$$d_B = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i-1 \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} qkoz. \qquad (12.21)$$

In the equation $a(i, j, k) = a(i, j-1, k)$ from system (12.3), we cannot directly recognise which is the calculated variable instance, and which is the used variable instance. This example elucidates the difference between *equations* and *assignments*. When fixing that $a(i, j, k)$ should follow from $a(i, j-1, k)$ by a ***copy operation***, we get the same dependence vector d_A as in (12.20). Correspondingly for the equation $b(i, j, k) = b(i-1, j, k)$.

A variable instance with iteration vector v is calculated in cell $P \cdot v$. If for this calculation another variable instance with iteration vector v' is needed, implying a data dependence with dependence vector $d = v - v'$, the used variable instance is provided by cell $P \cdot v'$. Therefore, we need a communication from cell $z' = P \cdot v'$ to cell $z = P \cdot v$. In systolic arrays, all communication has to be via direct static links between the communicating cells. Due to the linearity of the transformation from (12.9), we have $z - z' = P \cdot v - P \cdot v' = P \cdot (v - v') = P \cdot d$.

If $P \cdot d = \vec{0}$, communication happens exclusively inside the calculating cell, i.e., in time, only—and not in space. Passing values in time is via registers of the calculating cell.

Whereas for $P \cdot d \neq \vec{0}$, a communication between different cells is needed. Then a link along the ***flow direction*** $P \cdot d$ must be provided from/to all cells of the systolic array. The vector $-P \cdot d$, oriented in counter flow direction, leads from space point z to space point z'.

If there is more than one dependence vector d, we need an appropriate *link* for each of them at every cell. Take for example the formulas (12.19), (12.20), and (12.21) together with (12.11), then we get $P \cdot d_A = (-1, 1)$, $P \cdot d_B = (0, -1)$, and $P \cdot d_C = (1, 0)$. In Figure 12.3(a), terminating at every cell, we see three links corresponding to the various vectors $P \cdot d$. This results in a ***hexagonal communication topology***—instead of the ***orthogonal communication topology*** from the first example.

12.2.6. Inferring the structure of the cells

Now we apply the space-related techniques from section 12.2.5 to time-related questions. A variable instance with iteration vector v is calculated in timestep $\pi \cdot v$. If this calculation uses another variable instance with iteration vector v', the former had been calculated in timestep $\pi \cdot v'$. Hence communication corresponding to the dependence vector $d = v - v'$ must take exactly $\pi \cdot v - \pi \cdot v'$ timesteps.

Since (12.6) describes a linear map, we have $\pi \cdot v - \pi \cdot v' = \pi \cdot (v - v') = \pi \cdot d$. According to the systolic principle, each communication must involve at least one register. The dependence vectors d are fixed, and so the choice of a time vector π is constrained by

$$\pi \cdot d \geq 1 \ . \tag{12.22}$$

In case $P \cdot d = \vec{0}$, we must provide ***registers*** for stationary variables in all cells. But each register is overwritten with a new value in every timestep. Hence, if $\pi \cdot d \geq 2$, the old value must be carried on to a further register. Since this is repeated for $\pi \cdot d$ timesteps, the cell needs exactly $\pi \cdot d$ registers per stationary variable. The values of the stationary variable successively pass all these registers before eventually being used. If $P \cdot d \neq \vec{0}$, the transport of values analogously goes by $\pi \cdot d$ registers, though these are not required to belong all to the same cell.

For each dependence vector d, we thus need an appropriate number of registers. In Figure 12.3(b), we see three input ports at the cell, corresponding to the dependence vectors d_A, d_B, and d_C. Since for these we have $P \cdot d \neq \vec{0}$. Moreover, $\pi \cdot d = 1$ due to (12.7) and (12.4). Thus, we need one register per dependence vector. Finally, the regularity of system (12.3) forces three output ports for every cell, opposite to the corresponding input ports.

Good news: we can infer in general that each cell needs only a few registers, because the number of dependence vectors d is statically bounded with a system like (12.3), and for each of the dependence vectors the amount of registers $\pi \cdot d$ has a fixed and usually small value.

The three input and output ports at every cell now permit the use of three moving matrices. Very differently from Figure 12.1, a dot product $\sum_{k=1}^{4} a_{ik} \cdot b_{kj}$ here is not calculated within a single cell, but dispersed over the systolic array. As a prerequisite, we had to dissolve the sum into a sequence of single additions. We call this principle a ***distributed generic operator***.

Apart from the three input ports with their registers, and the three output ports, Figure 12.3(b) shows a multiplier chained to an adder. Both units are induced in each cell by applying the transformation (12.9) to the domain S of the equation $c(i, j, k) = c(i, j, k - 1) + a(i, j - 1, k) \cdot b(i - 1, j, k)$ from system (12.3). According to this equation, the addition has to follow the calculation of the product, so the order of the hardware operators as seen in Figure 12.3(b) is implied.

The source cell for each of the used operands follows from the projection of the corresponding dependence vector. Here, variable $a(i, j - 1, k)$ is related to the dependence vector $d_A = (0, 1, 0)$. The projection $P \cdot d_A = (-1, 1)$ constitutes the *flow direction* of matrix A. Thus the value to be used has to be expected, as observed by the calculating cell, in opposite direction $(1, -1)$, in this case from the port in the lower left corner of the cell, passing through register A. All the same, $b(i - 1, j, k)$ comes from the right via register B, and $c(i, j, k - 1)$ from above through register C. The calculated values $a(i, j, k)$, $b(i, j, k)$, and $c(i, j, k)$ are output into the opposite directions through the appropriate ports: to the upper right, to the left, and downwards.

If alternatively we use the projection matrix P from (12.8), then for d_C we get the direction $(0,0)$. The formula $\pi \cdot d_C = 1$ results in the requirement of exactly one register C for each item of the matrix C. This register provides the value $c(i,j,k-1)$ for the calculation of $c(i,j,k)$, and after this calculation receives the value $c(i,j,k)$. All this reasoning matches with the cell from Figure 12.1(b). Figure 12.1(a) correspondingly shows no links for matrix C between the cells: for the matrix is *stationary*.

Exercises

12.2-1 Each projection vector u induces several corresponding projection matrices P.

a. Show that
$$P = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix}$$
also is a projection matrix fitting with projection vector $u = (1,1,1)$.

b. Use this projection matrix to transform the domain from system (12.3).

c. The resulting space coordinates differ from that in section 12.2.3. Why, in spite of this, both point sets are topologically equivalent?

d. Analyse the cells in both arrangements for common and differing features.

12.2-2 Apply all techniques from section 12.2 to system (12.3), employing a space-time matrix
$$T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} .$$

12.3. Input/output schemes

In Figure 12.3(a), the *input/output scheme* is only sketched by the*flow directions* for the matrices A, B, C. The necessary details to understand the input/output operations are now provided by Figure 12.6.

The ***input/output scheme*** in Figure 12.6 shows some new phenomena when compared with Figure 12.1(a). The input and output cells belonging to any matrix are no longer threaded all on a single straight line; now, for each matrix, they lie along two adjacent *borders*, that additionally may differ in the number of links to the outside world. The data structures from Figure 12.6 also differ from that in Figure 12.1(a) in the angle of inclination. Moreover, the matrices A and B from Figure 12.6 arrive at the *boundary cells* with only one third of the *data rate*, compared to Figure 12.1(a).

Spending some effort, even here it might be possible in principle to construct—item by item—the appropriate input/output scheme fitting the present systolic array. But it is much more safe to apply a formal derivation. The following subsections are devoted to the presentation of the various methodical steps for achieving our goal.

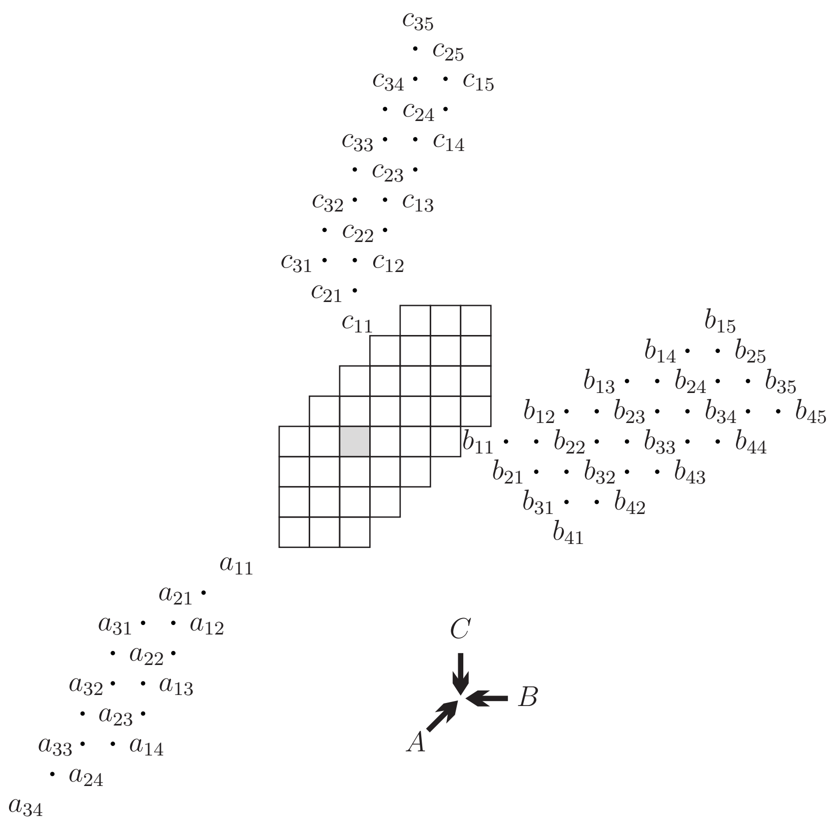


Figure 12.6. Detailed input/output scheme for the systolic array from Figure 12.3(a).

12.3.1. From data structure indices to iteration vectors

First, we need to construct a formal relation between the abstract data structures and the concrete variable instances in the assignment-free representation.

Each item of the matrix A can be characterised by a row index i and a column index k. These ***data structure indices*** can be comprised in a ***data structure vector*** $w = (i, k)$. Item a_{ik} in system (12.3) corresponds to the instances $a(i, j, k)$, with any j. The coordinates of these instances all lie on a line along direction $q = (0, 1, 0)$ in space $\mathbb{R}^3$. Thus, in this case, the formal change from data structure vector (i, k) to coordinates (i, j, k) can be described by the transformation

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ k \end{pmatrix} + j \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{12.23}$$

In system (12.3), the coordinate vector (i, j, k) of every variable instance equals the iteration vector of the domain point representing the calculation of this variable instance. Thus we also may interpret formula (12.23) as a relation between *data structure vectors* and *iteration vectors*. Abstractly, the desired iteration vectors v can be inferred from the data structure vector w by the formula

$$v = H \cdot w + \lambda \cdot q + p \; . \tag{12.24}$$

The affine vector p is necessary in more general cases, though always null in our example.

Because of $b(i, j, k) = b_{kj}$, the representation for matrix B correspondingly is

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} k \\ j \end{pmatrix} + i \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{12.25}$$

Concerning matrix C, each variable instance $c(i, j, k)$ may denote a different value. Nevertheless, all instances $c(i, j, k)$ to a fixed index pair (i, j) can be regarded as belonging to the same matrix item c_{ij}, since they all stem from the serialisation of the sum operator for the calculation of c_{ij}. Thus, for matrix C, following formula (12.24) we may set

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + k \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{12.26}$$

12.3.2. Snapshots of data structures

Each of the three matrices A, B, C is generated by two directions with regard to the *data structure indices*: along a row, and along a column. The difference vector $(0, 1)$ thereby describes a move from an item to the next item of the same row, i.e., in the next column: $(0, 1) = (x, y + 1) - (x, y)$. Correspondingly, the difference vector $(1, 0)$ stands for sliding from an item to the next item in the same column and next row: $(1, 0) = (x + 1, y) - (x, y)$.

Input/output schemes of the appearance shown in Figures 12.1(a) and 12.6 denote ***snapshots***: all positions of data items depicted, with respect to the entire systolic array, are related to a common timestep.

As we can notice from Figure 12.6, the rectangular shapes of the abstract data structures are mapped to parallelograms in the snapshot, due to the linearity of the applied space-time transformation. These parallelograms can be described by difference vectors along their borders, too.

Next we will translate difference vectors Δw from data structure vectors into spatial difference vectors Δz for the snapshot. Therefore, by choosing the parameter λ in formula (12.24), we pick a pair of iteration vectors v, v' that are mapped to the same timestep under our space-time transformation. For the moment it is not important which concrete timestep we thereby get. Thus, we set up

$$\pi \cdot v = \pi \cdot v' \quad \text{with} \quad v = H \cdot w + \lambda \cdot q + p \quad \text{and} \quad v' = H \cdot w' + \lambda' \cdot q + p \; , \tag{12.27}$$

implying

$$\pi \cdot H \cdot (w - w') + (\lambda - \lambda') \cdot \pi \cdot q = 0 \; , \tag{12.28}$$

and thus

$$\Delta\lambda = (\lambda - \lambda') = \frac{-\pi \cdot H \cdot (w - w')}{\pi \cdot q} \ . \qquad (12.29)$$

Due to the linearity of all used transformations, the wanted spatial difference vector Δz hence follows from the difference vector of the data structure $\Delta w = w - w'$ as

$$\Delta z = P \cdot \Delta v = P \cdot H \cdot \Delta w + \Delta\lambda \cdot P \cdot q \ , \qquad (12.30)$$

or

$$\Delta z = P \cdot H \cdot \Delta w - \frac{\pi \cdot H \cdot \Delta w}{\pi \cdot q} \cdot P \cdot q \ . \qquad (12.31)$$

With the aid of formula (12.31), we now can determine the spatial difference vectors Δz for matrix A. As mentioned above, we have

$$H = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, \quad q = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad P = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix}, \quad \pi = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}.$$

Noting $\pi \cdot q = 1$, we get

$$\Delta z = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \Delta w + \Delta\lambda \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \text{with} \quad \Delta\lambda = -\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \Delta w \ .$$

For the rows, we have the difference vector $\Delta w = (0, 1)$, yielding the spatial difference vector $\Delta z = (2, -1)$. Correspondingly, from $\Delta w = (1, 0)$ for the columns we get $\Delta z = (1, -2)$. If we check with Figure 12.6, we see that the rows of A in fact run along the vector $(2, -1)$, the columns along the vector $(1, -2)$.

Similarly, we get $\Delta z = (-1, 2)$ for the rows of B, and $\Delta z = (1, 1)$ for the columns of B; as well as $\Delta z = (-2, 1)$ for the rows of C, and $\Delta z = (-1, -1)$ for the columns of C.

Applying these instruments, we are now able to reliably generate appropriate input/output schemes—although separately for each matrix at the moment.

12.3.3. Superposition of input/output schemes

Now, the shapes of the matrices A, B, C for the snapshot have been fixed. But we still have to adjust the matrices relative to the systolic array—and thus, also relative to each other. Fortunately, there is a simple graphical method for doing the task.

We first choose an arbitrary iteration vector, say $v = (1, 1, 1)$. The latter we map with the projection matrix P to the cell where the calculation takes place,

$$z = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \ .$$

The iteration vector $(1, 1, 1)$ represents the calculations $a(1, 1, 1)$, $b(1, 1, 1)$, and $c(1, 1, 1)$; these in turn correspond to the data items a_{11}, b_{11}, and c_{11}. We now lay the input/output schemes for the matrices A, B, C on the systolic array in a way that the entries a_{11}, b_{11}, and c_{11} all are located in cell $z = (0, 0)$.

In principle, we would be done now. Unfortunately, our input/output schemes overlap with the cells of the systolic array, and are therefore not easily perceivable. Thus, we simultaneously retract the input/output schemes of all matrices in counter flow direction, place by place, until there is no more overlapping. With this method, we get exactly the input/output scheme from Figure 12.6.

As an alternative to this nice graphical method, we also could formally calculate an overlap-free placement of the various input/output schemes.

Only after specifying the input/output schemes, we can correctly calculate the number of timesteps effectively needed. The first relevant timestep starts with the first input operation. The last relevant timestep ends with the last output of a result. For the example, we determine from Figure 12.6 the beginning of the calculation with the input of the data item b_{11} in timestep 0, and the end of the calculation after output of the result c_{35} in timestep 14. Altogether, we identify 15 timesteps—five more than with pure treatment of the real calculations.

12.3.4. Data rates induced by space-time transformations

The input schemes of the matrices A and B from Figure 12.1(a) have a dense layout: if we drew the borders of the matrices shown in the Figure, there would be no spare places comprised.

Not so in Figure 12.6. In any input data stream, each data item is followed by two spare places there. For the input matrices this means: the *boundary cells* of the systolic array receive a proper data item only every third timestep.

This property is a direct result of the employed space-time transformation. In both examples, the abstract data structures themselves are dense. But how close the various items really come in the input/output scheme depends on the absolute value of the determinant of the transformation matrix T: in every input/output data stream, the proper items follow each other with a spacing of exactly $|\det(T)|$ places. Indeed $|\det(T)| = 1$ for Figure 12.1; as for Figure 12.6, we now can rate the fluffy spacing as a practical consequence of $|\det(T)| = 3$.

What to do with spare places as those in Figure 12.6? Although each cell of the systolic array from Figure 12.3 in fact does useful work only every third timestep, it would be nonsense to pause during two out of three timesteps. Strictly speaking, we can argue that values on places marked with dots in Figure 12.6 have no influence on the calculation of the shown items c_{ij}, because they never reach an active cell at time of the calculation of a variable $c(i, j, k)$. Thus, we may simply fill spare places with any value, no danger of disturbing the result. It is even feasible to execute three different matrix products at the same time on the systolic array from Figure 12.3, without interference. This will be our topic in section 12.3.7.

12.3.5. Input/output expansion and extended input/output scheme

When further studying Figure 12.6, we can identify another problem. Check, for example, the itinerary of c_{22} through the cells of the systolic array. According to the space-time transformation, the calculations contributing to the value of c_{22} happen in the cells $(-1, 0)$, $(0, 0)$, $(1, 0)$, and $(2, 0)$. But the input/output scheme from Figure 12.6 tells us that c_{22} also passes through cell $(-2, 0)$ before, and eventually visits cell $(3, 0)$, too.

This may be interpreted as some ***spurious calculations*** being introduced into the system

(12.3) by the used space-time transformation, here, for example, at the new domain points $(2, 2, 0)$ and $(2, 2, 5)$. The reason for this phenomenon is that the *domains of the input/output operations* are not in parallel to the chosen *projection direction*. Thus, some input/output operations are projected onto cells that do not belong to the ***boundary*** of the systolic array. But in the interior of the systolic array, no input/output operation can be performed directly. The problem can be solved by extending the trajectory, in flow or counter flow direction, from these inner cells up to the boundary of the systolic array. But thereby we introduce some new calculations, and possibly also some new domain points. This technique is called ***input/output expansion***.

We must avoid that the additional calculations taking place in the cells $(-2, 0)$ and $(3, 0)$ corrupt the correct value of c_{22}. For the matrix product, this is quite easy—though the general case is more difficult. The generic sum operator has a neutral element, namely zero. Thus, if we can guarantee that by new calculations only zero is added, there will be no harm. All we have to do is providing always at least one zero operand to any spurious multiplication; this can be achieved by filling appropriate input slots with zero items.

Figure 12.7 shows an example of a properly extended input/output scheme. Preceding and following the items of matrix A, the necessary zero items have been filled in. Since the entered zeroes count like data items, the input/output scheme from Figure 12.6 has been retracted again by one place. The calculation now begins already in timestep -1, but ends as before with timestep 14. Thus we need 16 timesteps altogether.

12.3.6. Coping with stationary variables

Let us come back to the example from Figure 12.1(a). For inputting the items of matrices A and B, no expansion is required, since these items are always used in *boundary cells* first. But not so with matrix C! The items of C are calculated in stationary variables, hence always in the same cell. Thus most results c_{ij} are produced in inner cells of the systolic array, from where they have to be moved—in a separate action—to *boundary cells* of the systolic array.

Although this new challenge, on the face of it, appears very similar to the problem from section 12.3.5, and thus very easy to solve, in fact we here have a completely different situation. It is not sufficient to extend existing data flows forward or backward up to the boundary of the systolic array. Since for stationary variables the dependence vector is the null vector, which constitutes no extensible direction, there can be no spatial flow induced by this dependency. Possibly, we can construct some auxiliary extraction paths, but usually there are many degrees of freedom. Moreover, we then need a *control mechanism* inside the cells. For all these reasons, the problem is further dwelled on in section 12.4.

12.3.7. Interleaving of calculations

As can be easily noticed, the *utilisation* of the systolic array from Figure 12.3 with input/output scheme from Figure 12.7 is quite poor. Even without any deeper study of the starting phase and the closing phase, we cannot ignore that the average utilisation of the array is below one third—after all, each cell at most in every third timestep makes a proper contribution to the calculation.

A simple technique to improve this behaviour is to ***interleave*** calculations. If we have three independent matrix products, we can successively input their respective data, delayed

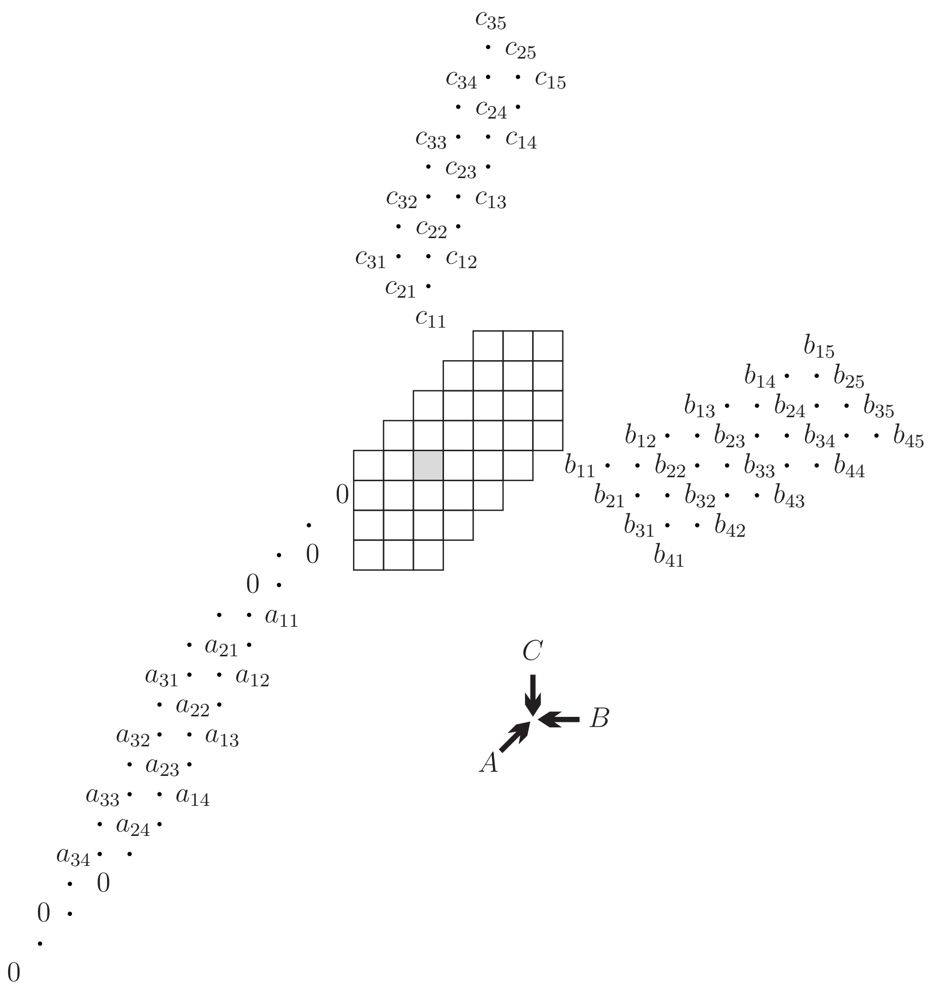


Figure 12.7. Extended input/output scheme, correcting Figure 12.6.

by only one timestep, without any changes to the systolic array or its cells. Figure 12.8 shows a snapshot of the systolic array, with parts of the corresponding input/output scheme.

Now we must check by a formal derivation whether this idea is really working. Therefore, we slightly modify system (12.3). We augment the variables and the domains by a fourth dimension, needed to distinguish the three matrix products:

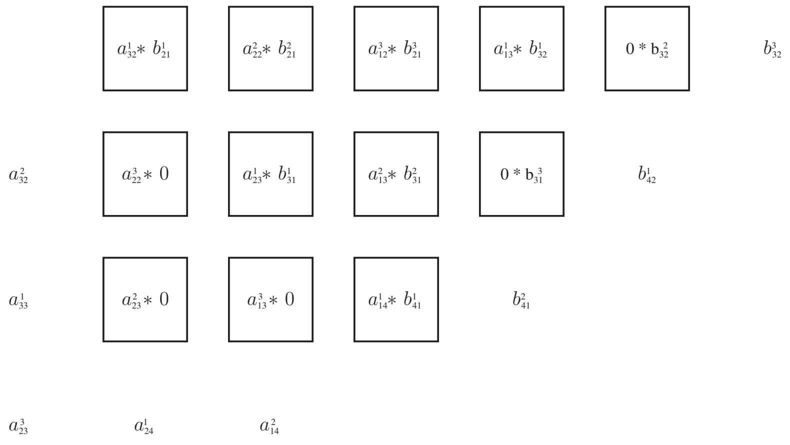


Figure 12.8. Interleaved calculation of three matrix products on the systolic array from Figure 12.3.

input operations

$$
\begin{array}{ll}
a(i,j,k,l) = a_{ik}^l & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, 1 \le l \le 3, \\
b(i,j,k,l) = b_{kj}^l & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3, \\
c(i,j,k,l) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0, 1 \le l \le 3.
\end{array}
$$

calculations and forwarding

$$
\begin{array}{ll}
a(i,j,k,l) = a(i,j-1,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3, \\
b(i,j,k,l) = b(i-1,j,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3, \\
c(i,j,k,l) = c(i,j,k-1,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3. \\
\quad + a(i,j-1,k,l) \cdot b(i-1,j,k,l) &
\end{array}
\tag{12.32}
$$

output operations

$$
\begin{array}{ll}
c_{ij}^l = c(i,j,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3, 1 \le l \le 3\ .
\end{array}
$$

Obviously, in system (12.32), problems with different values of l are not related. Now we must preserve this property in the systolic array. A suitable *space-time matrix* would be

$$
T = \begin{pmatrix} 0 & -1 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} . \tag{12.33}
$$

Notice that T is not square here. But for calculating the space coordinates, the fourth dimension of the iteration vector is completely irrelevant, and thus can simply be neutralised by corresponding zero entries in the fourth column of the first and second rows of T.

The last row of T again constitutes the *time vector* π. Appropriate choice of π embeds the three problems to solve into the space-time continuum, avoiding any intersection. Corresponding instances of the iteration vectors of the three problems are projected to the same

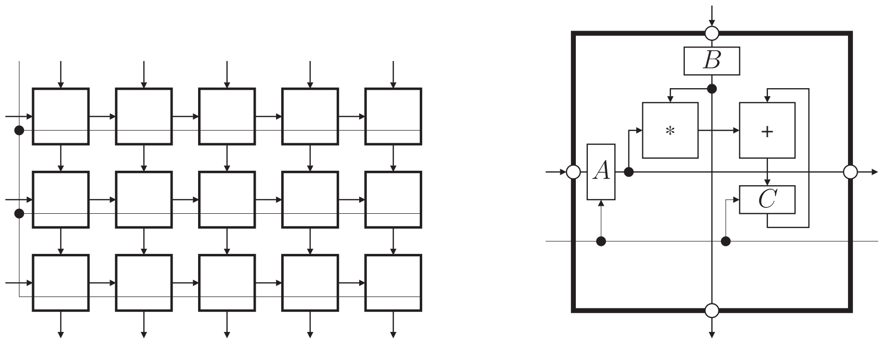


Figure 12.9. Resetting registers via global control. **(a)** Array structure. **(b)** Cell structure.

cell with a respective spacing of one timestep, because the fourth entry of π equals 1.

Finally, we calculate the average *utilisation*—with or without interleaving—for the concrete problem parameters $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$. For a single matrix product, we have to perform $N_1 \cdot N_2 \cdot N_3 = 60$ calculations, considering a multiplication and a corresponding addition as a compound operation, i.e., counting both together as only one calculation; input/output operations are not counted at all. The systolic array has 36 cells.

Without interleaving, our systolic array altogether takes 16 timesteps for calculating a single matrix product, resulting in an average utilisation of $60/(16 \cdot 36) \approx 0.104$ calculations per timestep and cell. When applying the described interleaving technique, the calculation of all three matrix products needs only two timesteps more, i.e., 18 timesteps altogether. But the number of calculations performed thereby has tripled, so we get an average utilisation of the cells amounting to $3 \cdot 60/(18 \cdot 36) \approx 0.278$ calculations per timestep and cell. Thus, by interleaving, we were able to improve the utilisation of the cells to 267 per cent!

Exercises

12.3-1 From equation (12.31), formally derive the spatial difference vectors of matrices B and C for the input/output scheme shown in Figure 12.6.

12.3-2 Augmenting Figure 12.6, draw an extended input/output scheme that forces both operands of all spurious multiplications to zero.

12.3-3 Apply the techniques presented in section 12.3 to the systolic array from Figure 12.1.

12.3-4★ Proof the properties claimed in section 12.3.7 for the special space-time transformation (12.33) with respect to system (12.32).

12.4. Control

So far we have assumed that each cell of a systolic array behaves in completely the same way during every timestep. Admittedly there are some relevant examples of such systolic arrays. However, in general the cells successively have to work in several ***operation modes***, switched to by some control mechanism. In the sequel, we study some typical situations for exerting control.

12.4.1. Cells without control

The cell from Figure 12.3(b) contains the registers A, B, and C, that—when activated by the global clock signal—accept the data applied to their inputs and then reliably reproduce these values at their outputs for one clock cycle. Apart from this system-wide activity, the function calculated by the cell is invariant for all timesteps: a *fused multiply-add* operation is applied to the three input operands A, B, and C, with result passed to a neighbouring cell; during the same cycle, the operands A and B are also forwarded to two other neighbouring cells. So in this case, the cell needs *no control* at all.

The *initial values* $c(i, j, 0)$ for the execution of the generic sum operator—which could also be different from zero here—are provided to the systolic array via the *input streams*, see Figure 12.7; the *final results* $c(i, j, N_3)$ continue to flow into the same direction up to the boundary of the array. Therefore, the input/output activities for the cell from Figure 12.3(b) constitute an intrinsic part of the normal cell function. The price to pay for this extremely simple cell function without any control is a restriction in all three dimensions of the matrices: on a systolic array like that from Figure 12.3, with *fixed* array parameters N_1, N_2, N_3, an $M_1 \times M_3$ matrix A can only be multiplied by an $M_3 \times M_2$ matrix B if the relations $M_1 \leq N_1$, $M_2 \leq N_2$, and $M_3 \leq N_3$ hold.

12.4.2. Global control

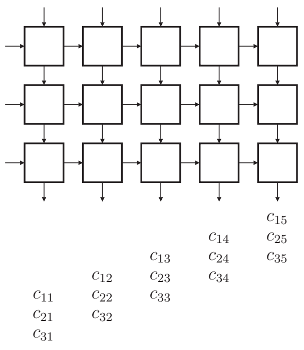
In this respect, constraints for the array from Figure 12.1 are not so restrictive: though the problem parameters M_1 and M_2 also are bounded by $M_1 \leq N_1$ and $M_2 \leq N_2$, there is no constraint for M_3. Problem parameters unconstrained in spite of fixed array parameters can only emerge in time but not in space, thus mandating the use of *stationary variables*.

Before a new calculation can start, each register assigned to a stationary variable has to be *reset* to an initial state independent from the previously performed calculations. For instance, concerning the systolic cell from Figure 12.3(b), this should be the case for register C. By a *global* signal similar to the clock, register C can be cleared in all cells at the same time, i.e., reset to a zero value. To prevent a corruption of the reset by the current values of A or B, at least one of the registers A or B must be *cleared* at the same time, too. Figure 12.9 shows an array structure and a cell structure implementing this idea.

12.4.3. Local control

Unfortunately, for the matrix product the principle of the global control is not sufficient without further measures. Since the systolic array presented in Figure 12.1 even lacks another essential property: the results c_{ij} are not passed to the boundary but stay in the cells.

At first sight, it seems quite simple to forward the results to the boundary: when the calculation of an item c_{ij} is finished, the links from cell (i, j) to the neighbouring cells $(i, j + 1)$ and $(i + 1, j)$ are no longer needed to forward items of the matrices A and B. These links can be reused then for any other purpose. For example, we could pass all items of C



c_{15}
c_{14} c_{25}
c_{13} c_{24} c_{35}
c_{12} c_{23} c_{34}
c_{11} c_{22} c_{33}
c_{21} c_{32}
c_{31}

Figure 12.10. Output scheme with delayed output of results.

through the downward-directed links to the lower border of the systolic array.

But it turns out that leading through results from the upper cells is hampered by ongoing calculations in the lower parts of the array. If the result c_{ij}, finished in timestep $i + j + N_3$, would be passed to cell $(i+1, j)$ in the next timestep, a conflict would be introduced between two values: since only one value per timestep can be sent from cell $(i + 1, j)$ via the lower port, we would be forced to keep either c_{ij} or $c_{i+1\,j}$, the result currently finished in cell $(i + 1, j)$. This effect would spread over all cells down.

To fix the problem, we could ***slow down*** the forwarding of items c_{ij}. If it would take two timesteps for c_{ij} to pass a cell, no collisions could occur. Then, the results stage a procession through the same link, each separated from the next by one timestep. From the lower boundary cell of a column, the host computer first receives the result of the bottom row, then that of the penultimate row; this procedure continues until eventually we see the result of the top row. Thus we get the output scheme shown in Figure 12.10.

How can a cell recognise when to change from forwarding items of matrix B to passing items of matrix C through the lower port? We can solve this task by an automaton combining global control with local control in the cell:

If we send a global signal to all cells at exactly the moment when the last items of A and B are input to cell $(1, 1)$, each cell can start a countdown process: in each successive timestep, we decrement a counter initially set to the number of the remaining calculation steps. Thereby cell (i, j) still has to perform $i + j - 1$ calculations before changing to *propagation mode*. Later, the already mentioned global reset signal switches the cell back to *calculation mode*.

Figure 12.11 presents a systolic array implementing this local/global principle. Basically, the array structure and the communication topology have been preserved. But each cell can run in one of two states now, switched by a *control logic*:

1. In *calculation mode*, as before, the result of the addition is written to register C. At the same time, the value in register B—i.e., the operand used for the multiplication—is forwarded through the lower port of the cell.
2. In *propagation mode*, registers B and C are connected in series. In this mode, the only function of the cell is to guide each value received at the upper port down to the lower port, thereby enforcing a *delay* of two timesteps.

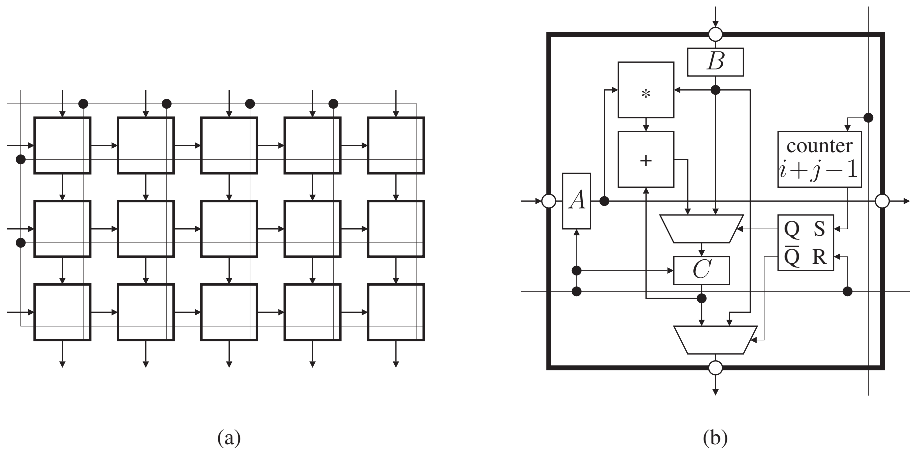


Figure 12.11. Combined local/global control. **(a)** Array structure. **(b)** Cell structure.

The first value output from cell (i, j) in *propagation mode* is the currently calculated value c_{ij}, stored in register C. All further output values are results forwarded from cells above. A formal description of the algorithm implemented in Figure 12.11 is given by the assignment-free system (12.34).

$$\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k) = a_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, \\
b(i,j,k) = b_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0. \\
\textit{calculations and forwarding} & \\
a(i,j,k) = a(i,j-1,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
b(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3. \\
\quad + a(i,j-1,k) \cdot b(i-1,j,k) & \\
\textit{propagation} & \\
b(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i + N_3, \\
c(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i - 1 + N_3, \\
\textit{output operations} & \\
c_{1+N_1+N_3-k,j} = b(i,j,k) & i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3 \ .
\end{array} \qquad (12.34)$$

It rests to explain how the control signals in a cell are generated in this model. As a prerequisite, the cell must contain a ***state flip-flop*** indicating the current *operation mode*.

The output of this flip-flop is connected to the control inputs of both multiplexors, see Figure 12.11(b). The global reset signal clears the state flip-flop, as well as the registers A and C: the cell now works in *calculation mode*.

The global ready signal starts the countdown in all cells, so in every timestep the counter is diminished by 1. The counter is initially set to the precalculated value $i + j - 1$, dependent on the position of the cell. When the counter reaches zero, the flip-flop is set: the cell switches to *propagation mode*.

If desisting from a direct reset of the register C, the last value passed, before the reset, from register B to register C of a cell can be used as a freely decidable initial value for the next dot product to evaluate in the cell. We then even calculate, as already in the systolic array from Figure 12.3, the more general problem

$$C = A \cdot B + D \, , \tag{12.35}$$

detailed by the following equation system:

$$\begin{array}{ll}
\textit{input operations} & \\
a(i,j,k) = a_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, \\
b(i,j,k) = b_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = d_{ij} & 1 \le i \le N_1, 1 \le j \le N_2, k = 0 \, . \\
\textit{calculations and forwarding} & \\
a(i,j,k) = a(i,j-1,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
b(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, \\
c(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \, . \\
\quad + a(i,j-1,k) \cdot b(i-1,j,k) & \\
\textit{propagation} & \\
b(i,j,k) = c(i,j,k-1) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i + N_3, \\
c(i,j,k) = b(i-1,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i - 1 + N_3 \, . \\
\textit{output operations} & \\
c_{1+N_1+N_3-k,j} = b(i,j,k) & i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3 \, .
\end{array} \tag{12.36}$$

12.4.4. Distributed control

The method sketched in Figure 12.11 still has the following drawbacks:

1. The systolic array uses global control signals, requiring a high technical accuracy.
2. Each cell needs a counter with counting register, introducing a considerable hardware expense.
3. The initial value of the counter varies between the cells. Thus, each cell must be individually designed and implemented.

4. The input data of any successive problem must wait outside the cells until all results from the current problem have left the systolic array.

These disadvantages can be avoided, if *control signals* are *propagated like data*—meaning a ***distributed control***. Therefore, we preserve the connections of the registers B and C with the multiplexors from Figure 12.11(b), but do not generate any control signals in the cells; also, there will be no global reset signal. Instead, a cell receives the necessary control signal from one of the neighbours, stores it in a new one-bit register S, and appropriately forwards it to further neighbouring cells. The primary control signals are generated by the *host computer*, and infused into the systolic array by *boundary cells*, only. Figure 12.12(a) shows the required array structure, Figure 12.12(b) the modified cell structure.

Switching to the *propagation mode* occurs successively down one cell in a column, always delayed by one timestep. The delay introduced by register S is therefore sufficient.

Reset to the *calculation mode* is performed via the same control wire, and thus also happens with a delay of one timestep per cell. But since the results c_{ij} sink down at half speed, only, we have to wait sufficiently long with the reset: if a cell is switched to *calculation mode* in timestep t, it goes to *propagation mode* in timestep $t + N_3$, and is reset back to *calculation mode* in timestep $t + N_1 + N_3$.

So we learned that in a systolic array, distributed control induces a different macroscopic timing behaviour than local/global control. Whereas the systolic array from Figure 12.12 can start the calculation of a new problem (12.35) every $N_1 + N_3$ timesteps, the systolic array from Figure 12.11 must wait for $2 \cdot N_1 + N_2 + N_3 - 2$ timesteps. The time difference $N_1 + N_3$ resp. $2 \cdot N_1 + N_2 + N_3 - 2$ is called the ***period***, its reciprocal being the ***throughput***.

System (12.37) formally describes the relations between distributed control and calculations. We thereby assume an infinite, densely packed sequence of matrix product problems, the additional iteration variable l being unbounded. The equation headed *variables with alias* describes but pure identity relations.

$$
\begin{array}{ll}
\textit{control} & \\
s(i,j,k,l) = 0 & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
s(i,j,k,l) = 1 & i = 0, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3, \\
s(i,j,k,l) = s(i-1,j,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ . \\[1em]
\textit{input operations} & \\
a(i,j,k,l) = a^l_{ik} & 1 \le i \le N_1, j = 0, 1 \le k \le N_3, \\
b(i,j,k,l) = b^l_{kj} & i = 0, 1 \le j \le N_2, 1 \le k \le N_3, \\
b(i,j,k,l) = d^{l+1}_{N_1+N_3+1-k,j} & i = 0, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ . \\[1em]
\textit{variables with alias} & \\
c(i,j,k,l) = c(i,j,N_1+N_3,l-1) & 1 \le i \le N_1, 1 \le j \le N_2, k = 0\ . \\[1em]
\textit{calculations and forwarding} & \\
a(i,j,k,l) = a(i,j-1,k,l) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ , \\
b(i,j,k,l) = \begin{cases} b(i-1,j,k,l) : s(i-1,j,k,l) = 0 \\ c(i,j,k-1,l) : s(i-1,j,k,l) = 1 \end{cases} & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ , \\
c(i,j,k,l) = \begin{cases} c(i,j,k-1,l) \\ \quad + a(i,j-1,k,l) \\ \quad \cdot b(i-1,j,k,l) : s(i-1,j,k,l) = 0 \\ b(i-1,j,k,l) \quad : s(i-1,j,k,l) = 1 \end{cases} & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ . \\[1em]
\textit{output operations} & \\
c^l_{1+N_1+N_3-k,j} = b(i,j,k,l) & i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ .
\end{array}
\tag{12.37}
$$

Formula (12.38) shows the corresponding space-time matrix. Note that one entry of T is not constant but depends on the *problem parameters*:

$$
T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & N_1 + N_3 \end{pmatrix} \tag{12.38}
$$

Interestingly, also the cells in a row switch one timestep later when moving one position to the right. Sacrificing some regularity, we could use this circumstance to relieve the *host computer* by applying control to the systolic array at cell $(1,1)$, only. We therefore would have to change the control scheme in the following way:

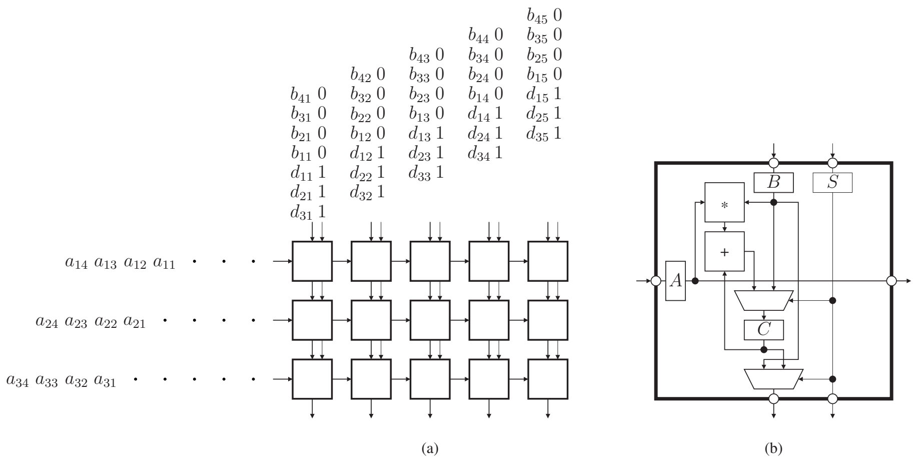

b_{41} 0 b_{31} 0 b_{21} 0 b_{11} 0 d_{11} 1 d_{21} 1 d_{31} 1
b_{42} 0 b_{32} 0 b_{22} 0 b_{12} 0 d_{12} 1 d_{22} 1 d_{32} 1
b_{43} 0 b_{33} 0 b_{23} 0 b_{13} 0 d_{13} 1 d_{23} 1 d_{33} 1
b_{44} 0 b_{34} 0 b_{24} 0 b_{14} 0 d_{14} 1 d_{24} 1 d_{34} 1
b_{45} 0 b_{35} 0 b_{25} 0 b_{15} 0 d_{15} 1 d_{25} 1 d_{35} 1
a_{14} a_{13} a_{12} a_{11}
a_{24} a_{23} a_{22} a_{21}
a_{34} a_{33} a_{32} a_{31}
(a)
B
S
*
+
A
C
(b)

Figure 12.12. Matrix product on a rectangular systolic array, with output of results and distributed control. **(a)** Array structure. **(b)** Cell structure.

$$
\begin{aligned}
&\textit{control}\\
&s(i,j,k,l) = 0 && i = 1, j = 0, 1 \le k \le N_3\ ,\\
&s(i,j,k,l) = 1 && i = 1, j = 0, 1 + N_3 \le k \le N_1 + N_3\ ,\\
&s(i,j,k,l) = s(i-1,j,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ ,\\
&\dots\\
&\textit{variables with alias}\\
&s(i,j,k,l) = s(i+1,j-1,k,l) && i = 0, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ ,\\
&\dots
\end{aligned}
\tag{12.39}
$$

Figure 12.13 shows the result of this modification. We now need cells of two kinds: cells on the upper border of the systolic array must be like that in Figure 12.13(b); all other cells would be as before, see Figure 12.13(c). Moreover, the *communication topology* on the upper border of the systolic array would be slightly different from that in the regular area.

12.4.5. The cell program as a local view

The chosen *space-time transformation* widely determines the architecture of the systolic array. Mapping recurrence equations to space-time coordinates yields an explicit view to the *geometric properties* of the systolic array, but gives no real insight into the *function of the cells*. In contrast, the processes performed inside a cell can be directly expressed by a ***cell program***. This approach is particularly of interest if dealing with a *programmable systolic array*, consisting of cells indeed controlled by a repetitive program.

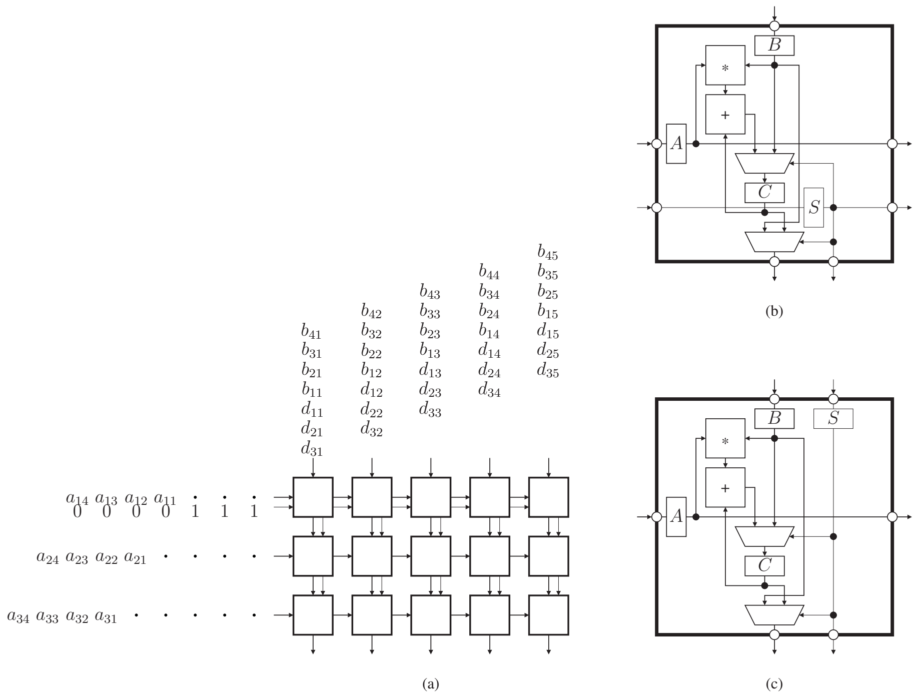

b_{41} b_{31} b_{21} b_{11} d_{11} d_{21} d_{31}
b_{42} b_{32} b_{22} b_{12} d_{12} d_{22} d_{32}
b_{43} b_{33} b_{23} b_{13} d_{13} d_{23} d_{33}
b_{44} b_{34} b_{24} b_{14} d_{14} d_{24} d_{34}
b_{45} b_{35} b_{25} b_{15} d_{15} d_{25} d_{35}
a_{14} a_{13} a_{12} a_{11} 0 0 0 0 1 1 1
a_{24} a_{23} a_{22} a_{21}
a_{34} a_{33} a_{32} a_{31}
(a)
B
*
+
A
C
S
(b)
(c)

Figure 12.13. Matrix product on a rectangular systolic array, with output of results and distributed control. **(a)** Array structure. **(b)** Cell on the upper border. **(c)** Regular cell.

Like the ***global view***, i.e., the *structure* of the systolic array, the ***local view*** given by a *cell program* in fact is already fixed by the space-time transformation. But, this local view is only induced implicitly here, and thus, by a further mathematical transformation, an explicit representation must be extracted, suitable as a cell program.

In general, we denote instances of program variables with the aid of ***index expressions***, that refer to iteration variables. Take, for instance, the equation

$$c(i, j, k) = c(i, j, k-1) + a(i, j-1, k) \cdot b(i-1, j, k) \quad 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3$$

from system (12.3). The instance $c(i, j, k-1)$ of the program variable c is specified using the index expressions i, j, and $k-1$, which can be regarded as functions of the iteration variables i, j, k.

As we have noticed, the set of iteration vectors (i, j, k) from the quantification becomes a set of space-time coordinates (x, y, t) when applying a space-time transformation (12.12) with transformation matrix T from (12.14),

$$\begin{pmatrix} x \\ y \\ t \end{pmatrix} = T \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix}. \tag{12.40}$$

Since each cell is denoted by space coordinates (x, y), and the cell program must refer to the current time t, the iteration variables i, j, k in the index expressions for the program variables are not suitable, and must be translated into the new coordinates x, y, t. Therefore, using the inverse of the space-time transformation from (12.40), we express the iteration variables i, j, k as functions of the space-time coordinates (x, y, t),

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \\ t \end{pmatrix} = \frac{1}{3} \cdot \begin{pmatrix} -1 & -2 & 1 \\ -1 & 1 & 1 \\ 2 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ t \end{pmatrix} . \tag{12.41}$$

The existence of such an inverse transformation is guaranteed if the space-time transformation is injective on the domain—and that it should always be: if not, some instances must be calculated by a cell in the same timestep. In the example, reversibility is guaranteed by the square, non singular matrix T, even without referral to the domain. With respect to the time vector π and any projection vector u, the property $\pi \cdot u \neq 0$ is sufficient.

Replacing iteration variables by space-time coordinates, which might be interpreted as a ***transformation of the domain***, frequently yields very unpleasant index expressions. Here, for example, from $c(i, j, k-1)$ we get

$$c((-x - 2 \cdot y + t)/3, (-x + y + t)/3, (2 \cdot x + y + t)/3)qkoz.$$

But, by a successive ***transformation of the index sets***, we can relabel the instances of the program variables such that the reference to cell and time appears more evident. In particular, it seems worthwhile to transform the equation system back into ***output normal form***, i.e., to denote the results calculated during timestep t in cell (x, y) by instances (x, y, t) of the program variables. We best gain a real understanding of this approach via an abstract mathematical formalism, that we can fit to our special situation.

Therefore, let

$$r(\psi_r(v)) = \mathcal{F}(\dots, s(\psi_s(v)), \dots) \qquad v \in S \tag{12.42}$$

be a quantified equation over a domain S, with program variables r and s. The ***index functions*** ψ_r and ψ_s generate the instances of the program variables as tuples of index expressions.

By transforming the domain with a function φ that is injective on S, equation (12.42) becomes

$$r(\psi_r(\varphi^{-1}(e))) = \mathcal{F}(\dots, s(\psi_s(\varphi^{-1}(e))), \dots) \qquad e \in \varphi(S), \tag{12.43}$$

where φ^{-1} is a function that constitutes an inverse of φ on $\varphi(S)$. The new index functions are $\psi_r \circ \varphi^{-1}$ and $\psi_s \circ \varphi^{-1}$.

Transformations of index sets don't touch the domain; they can be applied to each program variable separately, since only the instances of this program variable are renamed, and in a consistent way. With such renamings ϑ_r and ϑ_s, equation (12.43) becomes

$$r(\vartheta_r(\psi_r(\varphi^{-1}(e)))) = \mathcal{F}(\dots, s(\vartheta_s(\psi_s(\varphi^{-1}(e)))), \dots) \qquad e \in \varphi(S) \ . \tag{12.44}$$

If output normal form is desired, $\vartheta_r \circ \psi_r \circ \varphi^{-1}$ has to be the identity.

In the most simple case (as for our example), ψ_r is the identity, and ψ_s is an affine transformation of the form $\psi_s(v) = v - d$, with constant d—the already known *dependence vector*. ψ_r then can be represented in the same way, with $d = \vec{0}$. Transformation of the domains happens by the space-time transformation $\varphi(v) = T \cdot v$, with an invertible matrix T.

For all index transformations, we choose the same $\vartheta = \varphi$. Thus equation (12.44) becomes

$$r(e) = \mathcal{F}(\ldots, s(e - T \cdot d), \ldots) \qquad e \in T(S). \tag{12.45}$$

For the generation of a *cell program*, we have to know the following information for every timestep: the operation to perform, the source of the data, and the destination of the results—known from assembler programs as `opc`, `src`, `dst`.

The operation to perform (`opc`) follows directly from the function $\mathcal{F}$. For a cell with control, we must also find the timesteps when to perform this individual function $\mathcal{F}$. The set of these timesteps, as a function of the space coordinates, can be determined by projecting the set $T(S)$ onto the time axis; for general polyhedric S with the aid of a *Fourier–Motzkin elimination*, for example.

In system (12.45), we get a new dependence vector $T \cdot d$, consisting of two components: a (vectorial) spatial part, and a (scalar) timely part. The *spatial* part Δz, as a difference vector, specifies *which* neighbouring cell has calculated the operand. We directly can translate this information, concerning the input of operands to cell z, into a port specifier with port position $-\Delta z$, serving as the `src` operand of the instruction. In the same way, the cell calculating the operand, with position $z - \Delta z$, must write this value to a port with port position Δz, used as the `dst` operand in the instruction.

The *timely* part of $T \cdot d$ specifies, as a time difference Δt, *when* the calculation of the operand has been performed. If $\Delta t = 1$, this information is irrelevant, because the reading cell z always gets the output of the immediately preceding timestep from neighbouring cells. However, for $\Delta t > 1$, the value must be buffered for $\Delta t - 1$ timesteps, either by the *producer* cell $z - \Delta z$, or by the *consumer* cell z—or by both, sharing the burden. This need can be realised in the cell program, for example, with $\Delta t - 1$ copy instructions executed by the producer cell $z - \Delta z$, preserving the value of the operand until its final output from the cell by passing it through $\Delta t - 1$ registers.

Applying this method to system (12.37), with transformation matrix T as in (12.38), yields

$$\begin{aligned}
s(x, y, t) &= s(x-1, y, t-1) \\
a(x, y, t) &= a(x, y-1, t-1) \\
b(x, y, t) &= \begin{cases} b(x-1, y, t-1) & : s(x-1, y, t-1) = 0 \\ c(x, y, t-1) & : s(x-1, y, t-1) = 1 \end{cases} \\
c(x, y, t) &= \begin{cases} c(x, y, t-1) \\ \; + a(x, y-1, t-1) \\ \; \cdot b(x-1, y, t-1) & : s(x-1, y, t-1) = 0 \\ b(x-1, y, t-1) & : s(x-1, y, t-1) = 1 \end{cases}
\end{aligned} \tag{12.46}$$

The iteration variable l, being relevant only for the input/output scheme, can be set to a fixed value prior to the transformation. The corresponding cell program for the systolic array from Figure 12.12, performed once in every timestep, reads as follows:

Cell-Program

```
1  $S \leftarrow C(-1,0)(0)$
2  $A \leftarrow C(0,-1)$
3  $B \leftarrow C(-1,0)(1:N)$
4  $C(1,0)(0) \leftarrow S$
5  $C(0,1) \leftarrow A$
6  if $S = 1$
7     then $C(1,0)(1:N) \leftarrow C$
8          $C \leftarrow B$
9     else $C(1,0)(1:N) \leftarrow B$
10         $C \leftarrow C + A \cdot B$
```

The port specifiers stand for local input/output to/from the cell. For each, a pair of qualifiers is derived from the geometric position of the ports relative to the centre of the cell. Port $\mathtt{C}(0,-1)$ is situated on the left border of the cell, $\mathtt{C}(0,1)$ on the right border; $\mathtt{C}(-1,0)$ is above the centre, $\mathtt{C}(1,0)$ below. Each port specifier can be augmented by a bit range: $\mathtt{C}(-1,0)(0)$ stands for bit 0 of the port, only; $\mathtt{C}(-1,0)(1:N)$ denotes the bits 1 to N. The designations $\mathtt{A}$, $\mathtt{B}$, ... without port qualifiers stand for registers of the cell.

By application of matrix T from (12.13) to system (12.36), we get

$$
\begin{aligned}
a(x,y,t) &= a(x,y-1,t-1) && 1+x+y \le t \le x+y+N_3 \ , \\
b(x,y,t) &= b(x-1,y,t-1) && 1+x+y \le t \le x+y+N_3 \ , \\
c(x,y,t) &= c(x,y,t-1) && 1+x+y \le t \le x+y+N_3 \ , \\
&\quad + a(x,y-1,t-1)\cdot b(x-1,y,t-1) && \\
b(x,y,t) &= c(x,y,t-1) && x+y+1+N_3 \le t \le 2\cdot x+y+N_3 \ , \\
c(x,y,t) &= b(x-1,y,t-1) && x+y+1+N_3 \le t \le 2\cdot x+y-1+N_3 \ .
\end{aligned}
\tag{12.47}
$$

Now the advantages of distributed control become obvious. The cell program for (12.46) can be written with referral to the respective timestep t, only. And thus, we need no reaction to global control signals, no counting register, no counting operations, and no coding of the local cell coordinates.

Exercises

12.4-1 Specify appropriate input/output schemes for performing, on the systolic arrays presented in Figures 12.11 and 12.12, two evaluations of system (12.36) that follow each other closest in time.

12.4-2 How could we change the systolic array from Figure 12.12, to efficiently support the calculation of matrix products with parameters $M_1 < N_1$ or $M_2 < N_2$?

12.4-3 Write a cell program for the systolic array from Figure 12.3.

12.4-4⋆ Which throughput allows the systolic array from Figure 12.3 for the assumed values of N_1, N_2, N_3? Which for general N_1, N_2, N_3?

12.4-5⋆ Modify the systolic array from Figure 12.1 such that the results stored in stationary variables are output through additional links directed half right down, i.e., from cell (i, j) to cell $(i+1, j+1)$. Develop an assignment-free equation system functionally equivalent to system (12.36), that is compatible with the extended structure. How looks the resulting

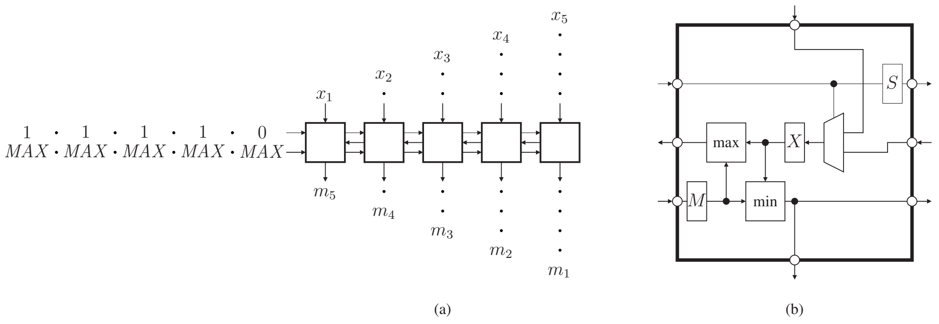


Figure 12.14. *Bubble sort* algorithm on a linear systolic array. **(a)** Array structure with input/output scheme. **(b)** Cell structure.

input/output scheme? Which period is obtained?

12.5. Linear systolic arrays

Explanations in the sections above heavily focused on *two-dimensional* systolic arrays, but in principle also apply to *one-dimensional* systolic arrays, called ***linear systolic arrays*** in the sequel. The most relevant difference between both kinds concerns the *boundary* of the systolic array. Linear systolic arrays can be regarded as consisting of *boundary cells*, only; under this assumption, input from and output to the *host computer* needs no special concern. However, the geometry of a linear systolic array provides one full dimension as well as one fictitious dimension, and thus communication along the full-dimensional axis may involve similar questions as in section 12.3.5. Eventually, the boundary of the linear systolic array can also be defined in a radically different way, namely to consist of both ***end cells***, only.

12.5.1. Matrix-vector product

If we set one of the problem parameters N_1 or N_2 to value 1 for a systolic array as that from Figure 12.1, the matrix product means to multiply a matrix by a vector, from left or right. The two-dimensional systolic array then ***degenerates*** to a one-dimensional systolic array. The vector by which to multiply is provided as an input data stream through an end cell of the linear systolic array. The matrix items are input to the array simultaneously, using the complete broadside.

As for full matrix product, results emerge stationary. But now, they either can be drained along the array to one of the end cells, or they are sent directly from the producer cells to the *host computer*. Both methods result in different control mechanisms, time schemes, and running time.

Now, would it be possible to provide *all* inputs via end cells? The answer is negative if the running time should be of complexity $\Theta(N)$. Matrix A contains $\Theta(N^2)$ items, thus there are $\Theta(N)$ items per timestep to read. But the number of items receivable through an end cell during one timestep is bounded. Thus, the ***input/output data rate***—of order $\Theta(N)$, here—may already constrain the possible design space.

12.5.2. Sorting algorithms

For sorting, the task is to bring the elements from a set $\{x_1, \ldots, x_N\}$, subset of a totally ordered basic set G, into an ascending order $\{m_i\}_{i=1,\ldots,N}$ where $m_i \le m_k$ for $i < k$. A solution to this problem is described by the following assignment-free equation system, where MAX denotes the maximum in G:

$$\begin{array}{ll}
\textit{input operations} & \\
x(i,j) = x_i & 1 \le i \le N, j = 0, \\
m(i,j) = MAX & 1 \le j \le N, i = j-1 . \\
& \\
\textit{calculations} & \\
m(i,j) = \min\{x(i,j-1), m(i-1,j)\} & 1 \le i \le N, 1 \le j \le i , \\
x(i,j) = \max\{x(i,j-1), m(i-1,j)\} & 1 \le i \le N, 1 \le j \le i . \\
& \\
\textit{output operations} & \\
m(i,j) = m_j & 1 \le j \le N, i = N .
\end{array} \tag{12.48}$$

By completing a projection along direction $u = (1, 1)$ to a space-time transformation

$$\begin{pmatrix} x \\ t \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}, \tag{12.49}$$

we get the linear systolic array from Figure 12.14, as an implementation of the *bubble sort* algorithm.

Correspondingly, the space-time matrix

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{12.50}$$

would induce another linear systolic array, that implements *insertion sort*. Eventually, the space-time matrix

$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{12.51}$$

would lead to still another linear systolic array, this one for *selection sort*.

For the sorting problem, we have $\Theta(N)$ input items, $\Theta(N)$ output items, and $\Theta(N)$ timesteps. This results in an input/output data rate of order $\Theta(1)$. In contrast to the matrix-vector product from section 12.5.1, the sorting problem with any prescribed input/output data rate in principle allows to perform the communication exclusively through the end cells of a linear systolic array.

Note that, in all three variants of sorting described so far, direct input is necessary to all cells: the values to order for bubble sort, the constant values *MAX* for insertion sort, and both for selection sort. However, instead of inputting the constants, the cells could generate them, or read them from a local memory.

All three variants require a cell control: insertion sort and selection sort use stationary variables; bubble sort has to switch between the processing of input data and the output of calculated values.

12.5.3. Lower triangular linear equation systems

System (12.52) below describes a *localised* algorithm for solving the linear equation system $A \cdot x = b$, where the $N \times N$ matrix A is a lower triangular matrix.

$$
\begin{array}{ll}
\textit{input operations} & \\
a(i,j) = a_{i,j+1} & 1 \le i \le N, 0 \le j \le i-1\ , \\
u(i,j) = b_i & 1 \le i \le N, j = 0\ . \\
& \\
\textit{calculations and forwarding} & \\
u(i,j) = u(i,j-1) - a(i,j-1) \cdot x(i-1,j) & 2 \le i \le N, 1 \le j \le i-1\ , \\
x(i,j) = u(i,j-1)/a(i,j-1) & 1 \le i \le N, j = i\ , \\
x(i,j) = x(i-1,j) & 2 \le i \le N-1, 1 \le j \le i-1\ . \\
& \\
\textit{output operations} & \\
x_i = x(i,j) & 1 \le i \le N, j = i\ .
\end{array}
\qquad (12.52)
$$

All previous examples had in common that, apart from copy operations, the same kind of calculation had to be performed for each domain point: fused multiply/add for the matrix algorithms, minimum and maximum for the sorting algorithms. In contrast, system (12.52) contains some domain points where multiply and subtract is required, as well as some others needing division.

When projecting system (12.52) to a linear systolic array, depending on the chosen *projection direction* we get fixed or varying cell functions. Peculiar for projecting along $u = (1,1)$, we see a single cell with divider; all other cells need a multiply/subtract unit. Projection along $u = (1,0)$ or $u = (0,1)$ yields identical cells, all containing a divider as well as a multiply/subtract unit. Projection vector $u = (1,-1)$ results in a linear systolic array with three different cell types: both end cells need a divider, only; all other cells contain a multiply/subtract unit, with or without divider, alternatingly. Thus, a certain projection can introduce ***inhomogeneities*** into a systolic array—that may be desirable, or not.

Exercises

12.5-1 For both variants of matrix-vector product as in section 12.5.1—output of the results by an end cell versus communication by all cells—specify a suitable array structure with input/output scheme and cell structure, including the necessary control mechanisms.

12.5-2 Study the effects of further projection directions on system (12.52).

12.5-3 Construct systolic arrays implementing insertion sort and selection sort, as mentioned in section 12.5.2. Also draw the corresponding cell structures.

12.5-4★ The systolic array for bubble sort from Figure 12.14 could be operated without control by cleverly organising the input streams. Can you find the trick?
12.5-5★ What purpose serves the value *MAX* in system (12.48)? How system (12.48) could be formulated without this constant value? Which consequences this would incur for the systolic arrays described?

Problems

12-1. Band matrix algorithms

In sections 12.1, 12.2, 12.5.1, and 12.5.3, we always assumed ***full*** input matrices, i.e., each matrix item a_{ij} used could be nonzero in principle. (Though in a lower triangular matrix, items above the main diagonal are all zero. Note, however, that these items are not inputs to any of the algorithms described.)

In contrast, practical problems frequently involve ***band matrices***, (see)cf. Kung/Leiserson [3]. In such a matrix, most diagonals are zero, left alone a small band around the main diagonal. Formally, we have $a_{ij} = 0$ for all i, j with $i - j \geq K$ or $j - i \geq L$, where K and L are positive integers. The ***band width***, i.e., the number of diagonals where nonzero items may appear, here amounts to $K + L - 1$.

Now the question arises whether we could profit from the band structure in one or more input matrices to optimise the systolic calculation. One opportunity would be to delete cells doing no useful work. Other benefits could be shorter input/output data streams, reduced running time, or higher throughput.

Study all systolic arrays presented in this chapter for improvements with respect to these criteria.

Chapter notes

The term *systolic array* has been coined by Kung and Leiserson in their seminal paper [3].

Karp, Miller, and Winograd did some pioneering work [2] for *uniform recurrence equations*.

Essential stimuli for a theory on the systematic design of systolic arrays have been Rao's PhD dissertation [6] and the work of Quinton [5].

The contribution of Teich and Thiele [8] shows that a formal derivation of the cell control can be achieved by methods very similar to those for a determination of the geometric array structure and the basic cell function.

The up-to-date book by Darte, Robert, and Vivien [1] joins advanced methods from compiler design and systolic array design, dealing also with the analysis of data dependences.

The monograph [9] still seems to be the most comprehensive work on systolic systems.

Each systolic array can also be modelled as a *cellular automaton*. The registers in a cell together hold the state of the cell. Thus, a *factorised* state space is adequate. Cells of different kind, for instance with varying cell functionality or position-dependent cell control, can be described with the aid of further components of the state space.

Each systolic algorithm also can be regarded as a *PRAM algorithm* with the same timing behaviour. Thereby, each register in a systolic cell corresponds to a PRAM memory cell, and vice versa. The EREW PRAM model is sufficient, because in every timestep exactly one systolic cell reads from this register, and then exactly one systolic cell writes to this register.

Each systolic system also is a special kind of *synchronous network* as defined by Lynch [4]. Time complexity measures agree. Communication complexity usually is no topic with systolic arrays. Restriction to input/output through boundary cells, frequently demanded for systolic arrays, also can be modelled in a synchronous network. The concept of *failures* is not required for systolic arrays.

The book [7] due to Sima, Kacsuk and Fountaine considers systolic systems in details.

Bibliography

[1] A. Darte, Y. Robert, F. Vivien. *Scheduling and Automatic Parallelization*. Birkhäuser Boston, 2000. 562

[2] R. M. Karp, R. E. Miller, S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967. 562

[3] H. T. Kung, C. E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff, G. W. Stewart (szerkesztők), *Sparse Matrix Proceedings, pp. 256–282*. SIAM, 1978. 562

[4] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publisher, 2001 (fifth edition). 563

[5] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture, pp. 208–214*, 1984. 562

[6] S. K. Rao. Regular iterative algorithms and their implementations on processor arrays. Doktori értekezés, Stanford University, 1985. 562

[7] D. Sima, T. Fountain, P. Kacsuk. *Advanced Computer Architectures: a Design Space Approach*. Addison-Wesley Publishing Company, 1998 (2. edition). 563

[8] J. Teich, L. Thiele. Control generation in the design of processor arrays. *International Journal of VLSI and Signal Processing*, 3(2):77–92, 1991. 562

[9] E. Zehendner. *Entwurf systolischer Systeme: Abbildung regulärer Algorithmen auf synchrone Prozessorarrays*. B. G. Teubner Verlagsgesellschaft, 1996. 562

Name index

Subject Index

Contents