

ECE 734 Project Report

Implementation of 2-D FIR filters on systolic arrays for image processing

Sebastian Siegel

Sebastian.Siegel@mailbox.tu-dresden.de

May 16, 2002

Instructor: Prof. Y.H. Hu

Abstract:

One major branch of image processing is filtering. It is capable of image enhancement, edge detection, noise reduction and other useful effects. As a 2-D FIR filter convolves an image with a filter matrix, the large amount of computations involved require a designated hardware to meet real time constraints. Fortunately, the regularity of the algorithm allows the usage of systolic arrays.

This project examines existing solutions and compares them to those that will be derived. The process of mapping an algorithm to a systolic array usually involves intermediate steps. After each step, certain decisions are necessary in terms of how to continue. It is one goal of this project to point out the consequences of each decision made as it may be a constraining force later on. Furthermore, the issue of symmetry will be addressed.

Contents

1. Introduction	4
2. Existing systolic arrays	5
2.1. A systolic 2-D convolution chip by H.T. Kung and S.W. Song	5
2.2. A New architecture for 2-D FIR digital filters by Z. Hu and G. King	6
3. Mapping the algorithm onto array structures	6
3.1. Data dependencies and Single Assignment Format	6
3.2. Mapping a dependence graph to a systolic array	9
3.3. Multiprojection	19
3.4. Symmetry	19
3.5. Partitioning	20
4. Conclusion	21
References	22
A. Appendix	23

List of Figures

1.1. Algorithm 1 (sequential execution)	4
2.1. Systolic Array according to Kung	5
2.2. Systolic Array according to Hu and King	6
3.1. Algorithm 2	7
3.2. Arranging the order of execution for the statement in the innermost loop	7
3.3. Dependence Graph 1	8
3.4. Space Transformation of Dependence Graph 1	10
3.5. Dependence Graph 3 (after Space Transformation)	11
3.6. Convex extension of Dependence Graph 3	13
3.7. Algorithm 3	14
3.8. Systolic Array 1	17
3.9. PE1	18
3.10. PE2	18
3.11. Systolic Array 2	19
3.12. PE3	20
3.13. Partitioning of the Image	21
A.1. Dependence Graph 2	24

List of Tables

1.	2D-FIR filter version 1	23
2.	2D-FIR filter version 2	25
3.	Systolic Array in Matlab	26
4.	2D-FIR filter version 3	27
5.	2D-FIR filter version 4	28

1. Introduction

2-D FIR image filters realized in the spatial domain convolve an image and a filter matrix according to formula (1.1) [1]:

$$f(i, j) = \sum_{v=-V}^V \sum_{u=-U}^U p(i+u, j+v) \cdot h(u, v) \quad (1.1)$$

where $U \leq i < I - U$, $V \leq j < J - V$ and the picture to be filtered ($p(i, j)$, $0 \leq i < I$, $0 \leq j < J$) is of size I (number of pixels vertically) and J (number of pixels horizontally). The filtered image is stored in a matrix $f(i, j)$. Its size is reduced by $2U$ rows and $2V$ columns in comparison to the original image $p(i, j)$. The filter matrix \mathbf{H} whose elements can be addressed as $h(u, v)$ consists of $2U + 1$ rows and $2V + 1$ columns.

As there are four different indices (i, j, u, v) in formula (1.1), a four level nested do-loop program¹ can be formulated to solve the task:

```

do i = U to I - 1 - U
  do j = V to J - 1 - V
    f(i, j) = 0
    do u = -U to U
      do v = -V to V
        f(i, j) = f(i, j) + p(i + u, j + v) · h(u, v)
      enddo v
    enddo u
  enddo j
enddo i

```

Figure 1.1: Algorithm 1 (sequential execution)

Formula (1.1) and algorithm 1 both show that there are $(I - 2U) \cdot (J - 2V) \cdot (2U + 1) \cdot (2V + 1)$ multiply-add (commonly realized on a MAC unit) operations to be performed until an image is filtered. Only symmetry² of the filter matrix \mathbf{H} could reduce the number of computations (let's assume that the picture holds no symmetry that could be of any use). A picture of the size 512 x 512 convolved with a 3 x 3 filter matrix therefore requires 2,340,900 MAC operations which take 234.09ms on a one processor machine running at 10MHz. Furthermore, algorithm 1 uses the value of almost every pixel several times. Without data reuse, the memory holding the picture would have to be addressed very often. I/O constraints usually do not allow that.

In order to perform filtering in real time applications, a multi-processor architecture, performing several computations in parallel, has to be derived and addressed with the task. Given the example above, 12 processors could perform the filtering at a rate of approx. 50 pictures/sec, if

¹see table 1 for a MatLab implementation of the algorithm

²see section 3.4 for details

their usage is around 100 percent. Due to the regularity of the algorithm, systolic arrays can be derived to speed up the computation. Existing architectures will be examined and compared to those derived in this project. The algorithms and systolic arrays derived are implemented in MatLab. Four functions (`FIR_2D_1.m` to `FIR_2D_4.m`) are available. Their usage is similar to `filter2.m`, the 2-D filter MatLab comes with.

2. Existing systolic arrays

2.1. A systolic 2-D convolution chip by H.T. Kung and S.W. Song

Kung and Song suggest in [2] a systolic array, which has a processor usage of 100 percent³. The systolic array can be used for filters where $U = V$. It consists of $(2U + 1)^3$ Processing Elements (PEs). The picture is evaluated row-wise, such that $4U + 1$ rows are worked with in parallel and $2U + 1$ rows of the filtered picture are computed. Therefore, $2U$ rows are read twice from the picture memory, for a 3×3 filter ($U = V = 1$) this will be 66.7% of the data.

As shown in figure 2.1, the systolic array consists of 3 kernel cells (3×3 PEs each) for a 3×3 filter. Each kernel cell needs an adder that can add the intermediate results of each row of PEs. The hardware is very specialized in a sense that it is dedicated to 3×3 filters and the total number of PEs is fixed. If these PEs cannot perform the filtering as fast as it may be required, additional kernel cells can be added. But the latency of the results of the last kernel cell increases significantly. The computing time is of the order $O[I^2 T_c / (2U + 1)]$, where the picture is of size $I \times I$, T_c is the cycle time of a basic cell and $V = U$.

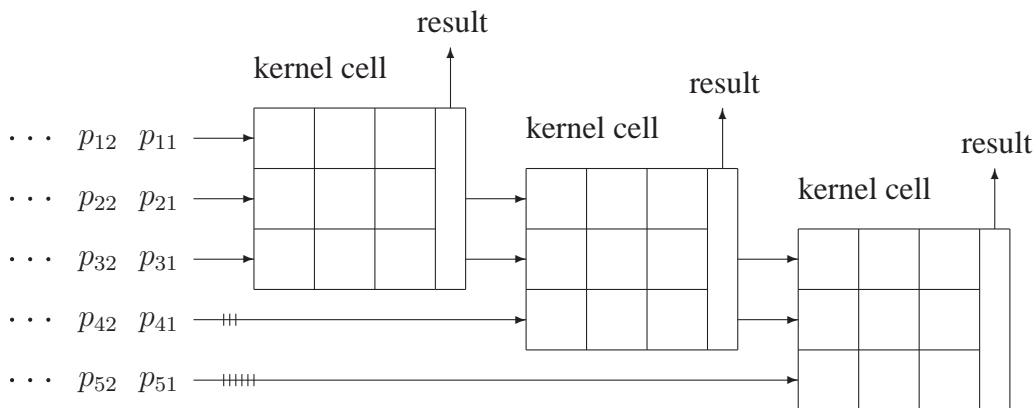


Figure 2.1: Systolic Array according to Kung

³apart from some lower efficiency at the beginning and at the end of each data block

2.2. A New architecture for 2-D FIR digital filters by Z. Hu and G. King

In [3] Hu and King suggest a fully pipelined architecture. Three pipelines interconnect all PEs locally. They propagate intermediate results, the input (pixels) and the filter coefficients. Their delays are pipeline are 1, 2, and 3 respectively. A draft of the architecture is shown in figure 2.2. It appears to be relatively similar to the architecture that will be derived later on, but the register usage is higher.

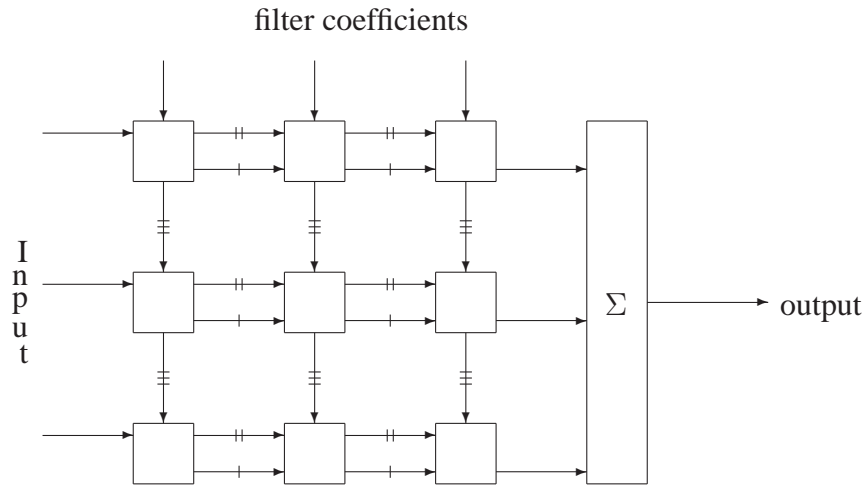


Figure 2.2: Systolic Array according to Hu and King

3. Mapping the algorithm onto array structures

3.1. Data dependencies and Single Assignment Format

In the formulation of algorithm 1 (see figure 1.1) $f(i, j)$ is both read and reassigned at the innermost loop. To avoid the dual usage of a storage capacity within one statement, the algorithm can be reformulated into a single assignment format (SAF) [4]. At the same time, the four level nested do-loop structure can be transformed into a three level nested do-loop structure by combining the indices u and v into n with a unique relation between pairs (u, v) and n . Furthermore, an intermediate variable f' is introduced to ensure that each variable is only assigned once during the execution of the algorithm. This results in algorithm 2.

In algorithm 1, an order of execution in terms of u and v for the innermost statement was fixed (see figure 3.2 (a) for an example). If this order is to be kept, the following relations between

```

do  $i = U$  to  $I - 1 - U$ 
  do  $j = V$  to  $J - 1 - V$ 
     $f'(i, j, 0) = 0$ 
    do  $n = 0$  to  $(2U + 1) \cdot (2V + 1) - 1$ 
       $f'(i, j, n + 1) = f'(i, j, n) + p(i + u(n), j + v(n)) \cdot h(u(n), v(n))$ 
    enddo  $n$ 
     $f(i, j) = f'(i, j, (2U + 1) \cdot (2V + 1))$ 
  enddo  $j$ 
enddo  $i$ 

```

Figure 3.1: Algorithm 2

u , v and n can be formulated (% represents a modulo division):

$$n = n(u, v) = (2V + 1) \cdot (u + U) + v + V \quad (3.1)$$

$$u = u(n) = \left\lfloor \frac{n}{2V + 1} \right\rfloor - U \quad (3.2)$$

$$v = v(n) = n \% (2V + 1) - V \quad (3.3)$$

As summands are interchangeable, the restrictions regarding the order of execution for the innermost statement in algorithm 1 are not necessary. Therefore, relations other than those mentioned in equations (3.1) to (3.3) ought to be considered. Two such examples are shown in figure 3.2 (b) and (c).

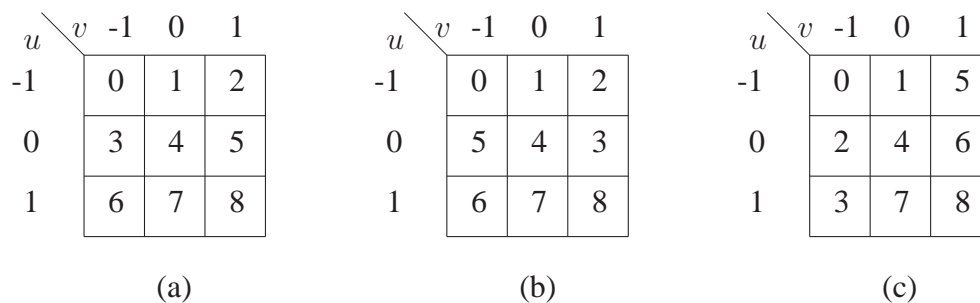


Figure 3.2: Examples of different orders of execution of the two innermost loops of algorithm 1 for $U = 1$ and $V = 1$ (n refers to the sequential execution of the innermost statement in algorithm 2 and it is shown within the grid. It could be a time index.)

A dependence graph (DG) [4] that corresponds to the single assignment format of algorithm 2 shows the data dependencies. Figure 3.3 gives an example of such a DG⁴ for the order of execution given in equations (3.1) to (3.3).

⁴to ensure that the dependence graph is readable, only some exemplary dependence vectors are drawn

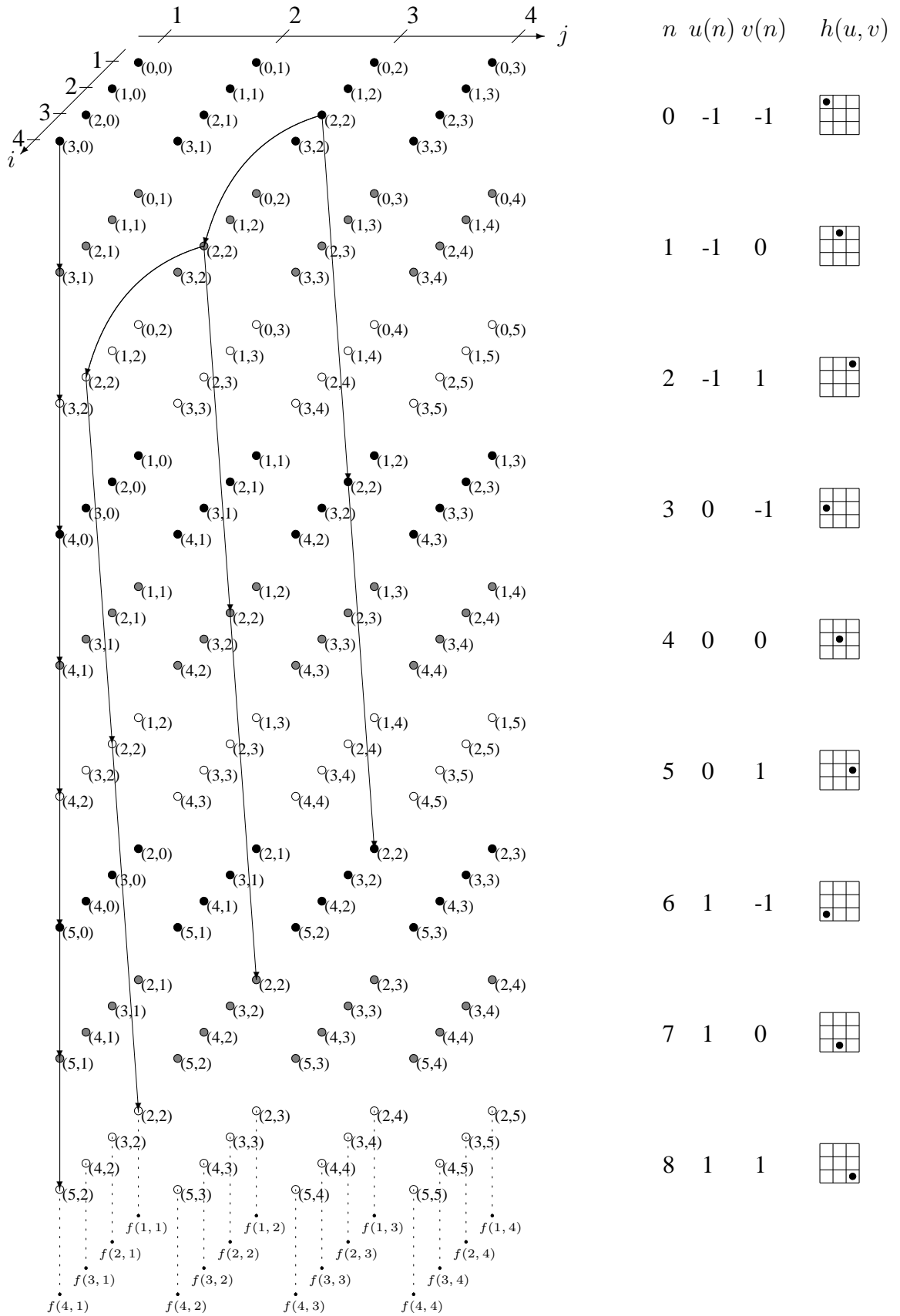


Figure 3.3: Dependence Graph 1 ($U = 1$ and $V = 1$). Note that only some dependence vectors are drawn to avoid confusion and that i and j go up to $I - 2$ and $J - 2$ respectively which is not shown in the graph.

According to algorithm 2, there is only one data dependency, that is vector $(0 \ 0 \ 1)^T$. To give an example, these vectors are drawn in figure 3.3 for the computation of $f(4, 1)$. All other data are assumed to be broadcasted, i.e. p and h . Direct access to processing elements (PEs) other than those bordering an array of PEs should be minimized or even avoided. That is because a systolic array typically consists of uniform and fully pipelined PEs, i.e. the whole system contains only local interconnections [4]. Therefore, the broadcasting of variables p and h should be changed into propagation. For $p(2, 2)$, an example of such an introduced data dependency is shown in figure 3.3. It is assumed that the value of $p(2, 2)$ enters the dependence graph at the uppermost node. From there on, it is made available to all nodes in need. This concept introduces two more dependence vectors: $(0 \ -1 \ 1)^T$ and $(-1 \ 0 \ 2V + 1)^T$.

The propagation of the pixels is possible for all p . They would enter the dependence graph at the uppermost layer⁵ ($n = 0$) and be propagated using local connections (these are the data dependencies most recently introduced). The values of h are constant for each layer. Therefore, the value $h(u(n), v(n))$ could enter layer n on one of four sides ($i = U, i = I - 1 - U, j = V$ or $j = J - 1 - V$) and be propagated through the layer using other local interconnections. One possibility would be such that all $h(u(n), v(n))$ enter layer n at nodes $i = U$ (meaning that $J - 2V$ copies of the value would have to be available). Then these values are propagated along the i direction, introducing another dependence vector: $(1 \ 0 \ 0)^T$.

As mentioned earlier, relations between (u, v) and n other than those shown in equations (3.1) to (3.3) are possible. For the relation shown in figure 3.2 (b), a different set of dependence vectors for the propagation of p results. Figure A.1 shows the resulting dependence vectors: $(0 \ -1 \ 1)^T$, $(-1 \ 0 \ 1)^T$ and $(0 \ 1 \ 1)^T$. A different order may therefore cause other data dependencies. As there are many different arrangements possible, only a limited number can be further regarded in detail. Section 3.2 will discuss what DGs should be kept and the impact of different dependence vectors.

3.2. Mapping a dependence graph to a systolic array

If a dependence graph consists of only two dimensions, a direct implementation as a systolic array may be possible, especially after the application of certain techniques such as retiming [4]. All DGs derived from algorithm 2 will be three dimensional and must therefore be mapped to a two dimensional graph, i.e. a systolic array in order to allow their implementation. Linear mapping [4] maps a D dimensional graph to a $D - 1$ dimensional graph by introducing a projection direction (along the projection vector \underline{d}) and a schedule for the nodes (represented by the scheduling vector \underline{s}).

Due to the projection, each node of the dependence graph will be assigned to a processor (in time). A processor base \mathbf{P} will span the processor space which is perpendicular to \underline{d} . After projection, the input and output nodes should be assigned to processors along the

⁵Note that the dependence graph would consist of I by J by $(2U + 1) \cdot (2V + 1)$ nodes. Some dummy nodes to pass data along would be necessary. The filtered picture is available at nodes $(i, j, (2U + 1) \cdot (2V + 1) - 1)$, where $U \leq i < I - U$ and $V \leq j < J - V$.

border of the systolic array in order to facilitate the VLSI design. As mentioned in section 3.1, all inputs (except for the filter coefficients h) can be made available at the uppermost layer of the dependence graph ($n = 0$). The output is available at the bottommost layer ($n = (U + 1) \cdot (V + 1) - 1$). Only a projection in the i direction and/or the j direction results in a systolic array where the border processors perform the I/O tasks. That means however, that the systolic array will be at least of size $(U + 1) \cdot (V + 1)$ by $\min\{I - 2U, J - 2V\}$. For $U = V = 2$ and $I = J = 512$, this systolic array would be at least of size 25 by 508. In order to realize it on a VLSI chip, partitioning schemes have to be applied. That introduces an extra tradeoff between I/O bandwidth and memory access.

Another approach is an early partitioning of the dependence graph, which – after reassembling the partitions – shows other properties. The local data dependencies reach no further than $2U + 1$ in the i direction and $2V + 1$ in the j direction for the pixel values p . Therefore, partitions of $2U + 1$ nodes for the i direction can be moved as shown in figure 3.4. The resulting dependence graph is shown in figure 3.5.

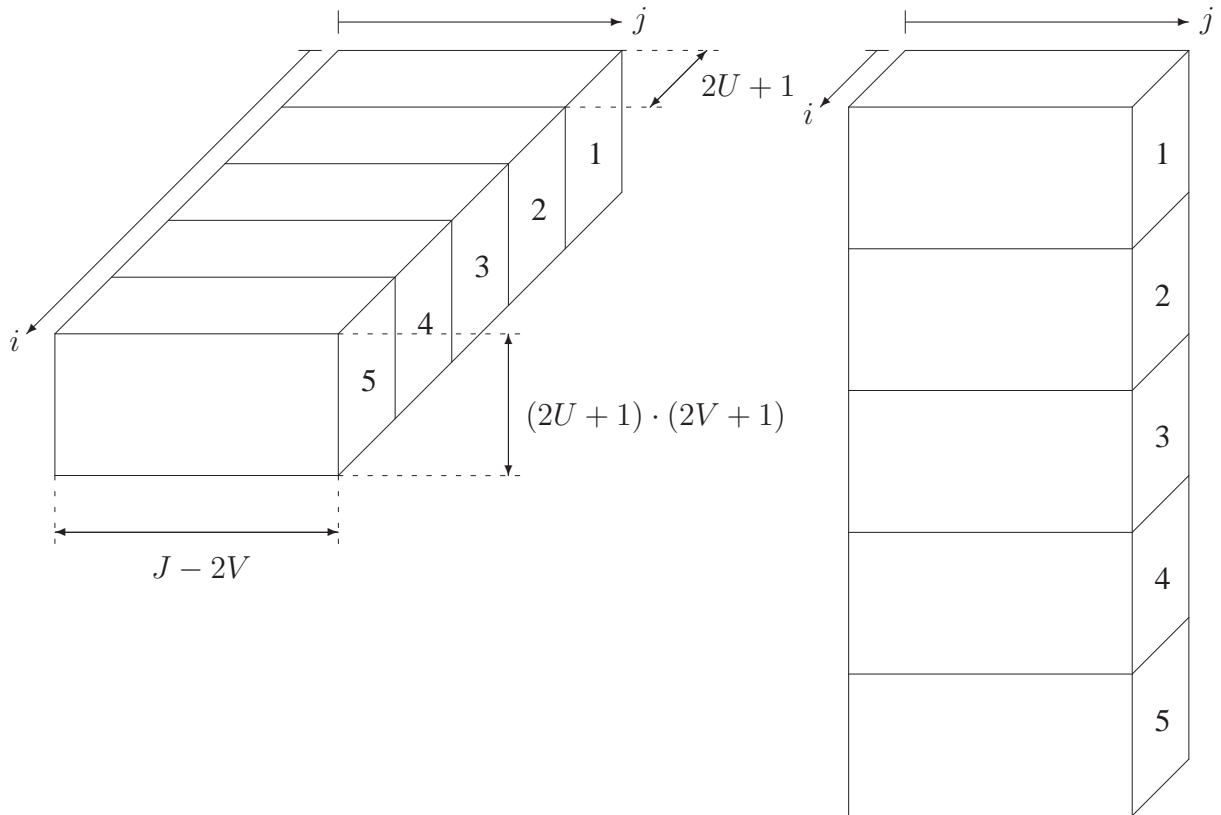


Figure 3.4: Space Transformation of Dependence Graph 1

Due to the regularity of the propagation of the pixel values p , there are no new dependence

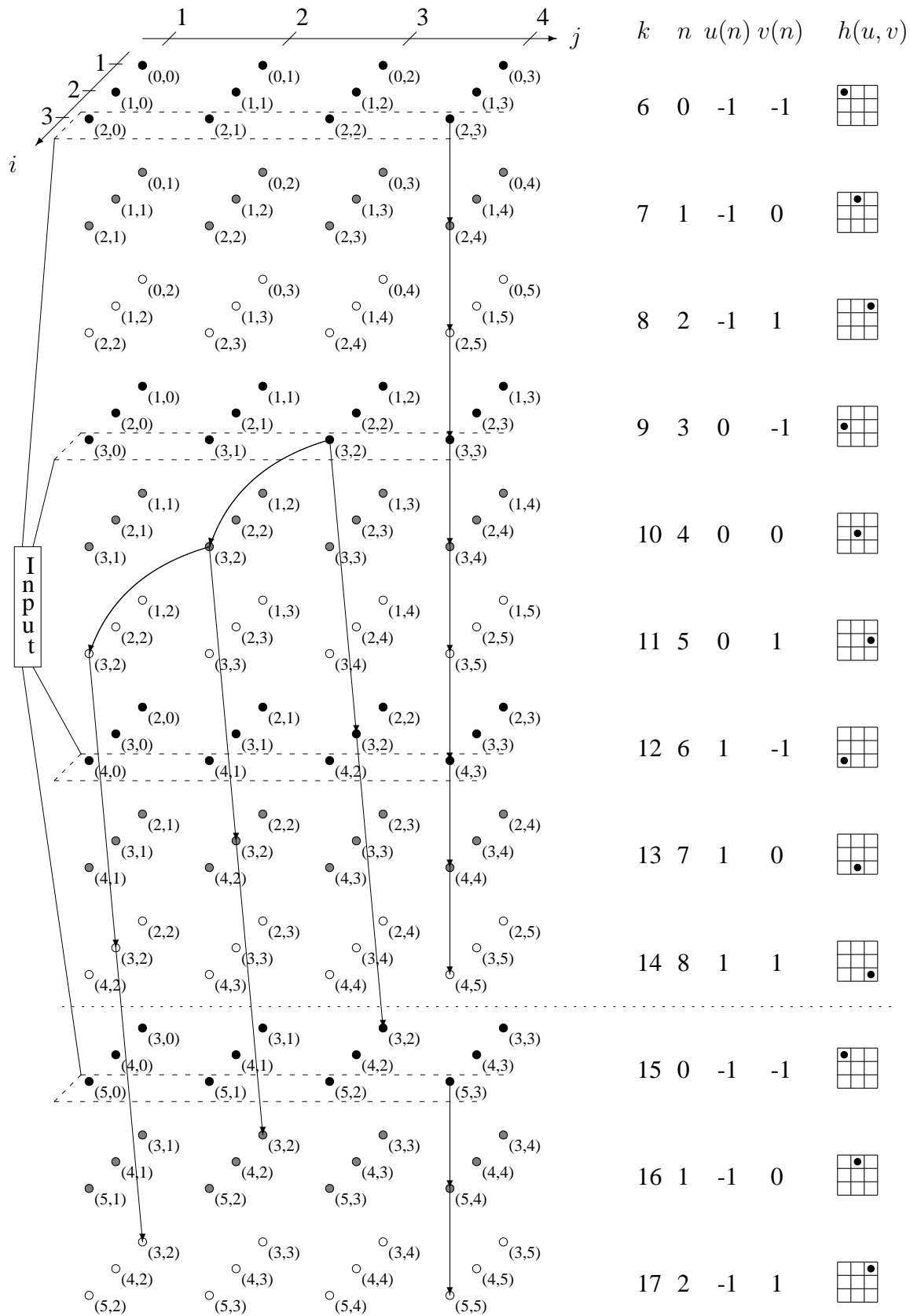


Figure 3.5: Dependence Graph 3 ($U = 1$ and $V = 1$). Note that only the dependence vectors that propagate $p(3, 2)$ and those that pipeline the partial sums of $f(3, 4)$ and two of $f(6, 4)$ are drawn. Further: j and k go up to $J - 1 + V = J$ and K respectively which is not shown in the graph.

vectors necessary to move the data from one partition⁶ to another. As it is shown in figure 3.5, the pixel values p that enter the dependence graph can be propagated in such a way that they enter a new partition as if there were no border between them (which there really is not). The input nodes for the pixel values p are now all nodes with the coordinates $(2U + 1, j, k)$, where $V \leq j < J + V$ and k has to represent a layer for which $v = -V$ is true. k is a new variable to distinguish multiple partitions. So far, the direction orthogonal to the i - j plane was spanned by n . Due to the reordering of the nodes, each value of n occurs several times, while k will be a variable counting all layers starting from zero.

Restricting the inputs to the nodes mentioned above results in a dependence graph with additional nodes (not shown in figure 3.5), which only perform data propagation. There are $2U \cdot (2V + 1)$ additional layers required so that the first $2U$ rows of pixel values can enter the dependence graph as well. These layers will be the uppermost in the dependence graph. That also explains why the uppermost layer shown in figure 3.5 is assigned $k = 6$. There are actually 6 layers above that one (not shown), so that the $p(0, \circ)$ and the $p(1, \circ)$ can enter the dependence graph as well.

The filtered picture (the values f) is available at all nodes where $u = U$ and $v = V$. That is, when $k \% ((2U + 1) \cdot (2V + 1)) = 2U \cdot (2V + 1) - 1$. The maximum k is denoted to be K .

$$K = 2U \cdot (2V + 1) - 1 + (2U + 1) \cdot (2V + 1) \cdot \left\lceil \frac{I - 2U}{2U + 1} \right\rceil$$

Example: $I = J = 16, U = 1, V = 2$:

Input nodes: $k = 0, 5, 10, \dots, 75$ and $i = 3$

Output nodes: $k = 9, 24, 39, \dots, K$

$$K = 2 \cdot 5 - 1 + 3 \cdot 5 \cdot \lceil 14/3 \rceil = 84$$

In the example above, the input nodes with $k = 80$ do not have to be assigned. As the result will have 14 rows ($I - 2U$), only the results at nodes $k = 84$ and $i = 1, 2$ are to be used for the filtered picture. The other 12 rows will be available in sets of three at the other layers denoted as output layers. That is also shown in figure 3.6.

Finally, algorithm 2 (see figure 3.1) can be rewritten in a single assignment format with localized data propagation for the values of p as well. That is shown in figure 3.7 (Algorithm 3)⁷.

The mapping is subject to certain constraints [4]:

$$\underline{s}^T \underline{v}_i > 0 \quad \text{causality} \quad (3.4)$$

$$\underline{s}^T \underline{d} \neq 0 \quad \text{to avoid resource conflicts} \quad (3.5)$$

The vector $\underline{s}^T = [s_1 \ s_2 \ s_3]^T$ is the scheduling vector which points perpendicular on hyper planes along which all nodes are executed simultaneously. There are four dependence vectors

⁶these partitions are shown in figure 3.4

⁷see also table 5 for a listing of a MatLab implementation

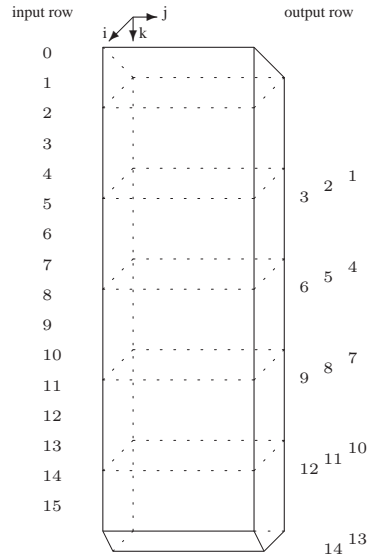


Figure 3.6: Convex extension of Dependence Graph 3 ($I = J = 16, U = 1, V = 2$)

$\underline{v}_i, i = 1, 2, 3, 4$ (\underline{v}_1 and \underline{v}_2 propagate the pixel values, \underline{v}_3 moves the partial sums of f through the dependence graph and \underline{v}_4 propagates the filter coefficients h):

$$\mathbf{D} = [\underline{v}_1 \ \underline{v}_2 \ \underline{v}_3 \ \underline{v}_4] = \begin{bmatrix} 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 1 & 2V + 1 & 1 & 0 \end{bmatrix}$$

The projection vector \underline{d} is chosen to be $[0 \ 0 \ 1]^T$ in order to minimize the size of the resulting systolic array. As $\underline{v}_3 = \underline{d}$, the constraint in formula (3.4) will be sufficient. With $\underline{d} = [0 \ 0 \ 1]^T$, the processor base is chosen to be

$$\mathbf{P}^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

A number of scheduling vectors \underline{s} meeting (3.4) can be generated. The multiplication $\underline{s}^T \underline{v}_i$ will also determine the number of registers on the resulting pipeline i . Each column of PEs ($j=\text{constant}$) should have a minimum number of registers. \underline{v}_1 will only occur once within each PE column (at PE with $i = 3U$). For \underline{v}_2 , there will be $2U$ pipelines connecting the $2U + 1$ PEs along such a column. \underline{v}_3 is a self loop after mapping. And the fourth dependence vector (\underline{v}_4) is not that important right now as it only pipelines the filter coefficients, which could also be broadcasted along each row of PEs (no registers needed). Therefore, the following expression – determining the registers necessary in each column of PEs – should be minimized:

```


$$K = 2U \cdot (2V + 1) - 1 + (2U + 1) \cdot (2V + 1) \cdot \lceil \frac{I-2U}{2U+1} \rceil$$

do  $k = 0$  to  $K$ 
   $u = \lfloor \frac{k-2U \cdot (2V+1)}{2V+1} \rfloor - U$ 
   $v = \lfloor ((k - 2U \cdot (2V + 1)) \% ((2U + 1) \cdot (2V + 1))) \% (2V + 1) - V \rfloor$ 
  do  $i = 3U$  downto  $U$ 
    do  $j = J - 1 + V$  downto  $V$ 
      if  $i = 3U$  and  $k \% (2V + 1) = 0$  and  $k < I \cdot (2V + 1)$  then
         $p'(i, j, k) = p(k / (2V + 1), j - V)$ 
      elseif  $j < J - 1 + V$ 
         $p'(i, j, k) = p'(i, j + 1, k - 1)$ 
      else
         $p'(i, j, k) = 0$ 
      end if
      if  $i < 3U$  and  $k \geq (3U - i) \cdot (2V + 1)$  then
         $p'(i, j, k) = p'(i + 1, j, k - (2V + 1))$ 
      end if
      if  $k \% ((2U + 1) \cdot (2V + 1)) = 2U \cdot (2V + 1)$  then
         $f'(i, j, k) = p'(i, j, k) \cdot h(u, v)$ 
      elseif  $k > 2U \cdot (2V + 1)$ 
         $f'(i, j, k) = f'(i, j, k - 1) + p'(i, j, k) \cdot h(u, v)$ 
      end if
      if  $k \% ((2U + 1) \cdot (2V + 1)) = 2U \cdot (2V + 1) - 1$  and
         $k \geq 2 \cdot (2V + 1) + (2U + 1) \cdot (2V + 1) - 1$  and  $j < J - V$  then
        if  $k < K$  or  $i < U + \lceil I - 2U - 0.5 \% (2U + 1) \rceil$ 
           $f(i + (k - (4U + 1) \cdot (2V + 1) + 1) / ((2U + 1) \cdot (2V + 1)), j) = f'(i, j, k)$ 
        end if
      end if
    enddo  $j$ 
  enddo  $i$ 
enddo  $k$ 

```

Figure 3.7: Algorithm 3

$$\begin{aligned}
S &= \underline{s}^T \underline{v}_1 + \underline{s}^T \underline{v}_2 \cdot 2U + \underline{s}^T \underline{v}_3 \cdot (2U + 1) \\
&= -s_2 + s_3 + (-s_1 + (2V + 1)s_3) \cdot 2U + s_3 \cdot (2U + 1) \\
&= -s_1 - s_2 + (4U(V + 1) + 2)s_3
\end{aligned} \tag{3.6}$$

At the same time (3.4) gives the following restrictions:

$$-s_2 + s_3 > 0 \tag{3.7}$$

$$-s_1 + (2V + 1)s_3 > 0 \tag{3.8}$$

$$s_3 > 0 \tag{3.9}$$

In order to minimize S from (3.6) and meet (3.9) at the same time, s_3 should be as small as possible and it is therefore chosen to be 1. s_1 should be as large as possible in order to minimize S . But it can only be as large as $2V$ in order to meet (3.8). Given s_3 , the only solution for s_2 is 0. Therefore:

$$\underline{s}^T = [2V \ 0 \ 1] \tag{3.10}$$

Another important fact is the total execution time T_{comp} which should be minimized as well. The following formula calculates T_{comp} [4].

$$T_{comp} = \max_{p,q \in DG} |\underline{s}^T(p - q)| + 1 \tag{3.11}$$

The DG consists of a convex region of nodes bound by 8 points⁸

$$\begin{aligned}
&(3U, V, 0), \quad (3U, V + J - 1, 0), \\
&(U, V, 2U(2V + 1)), \quad (U, V + J - 1, 2U(2V + 1)), \\
&(U, V, K), \quad (U, V + J - 1, K), \\
&(U + I\%(2U + 1), V, K), \quad (U + I\%(2U + 1), V + J - 1, K)
\end{aligned}$$

One way to minimize T_{comp} is to chose \underline{s}^T to be $[0 \ 1 \ 0]$, in which case $T_{comp} = J$. According to (3.4), this is not permissible. From (3.9) we know that $s_3 > 0$. Therefore, the s_3 -component in (3.11) will contribute a major part: K . s_2 had to be zero, so T_{comp} will not be affected by it. Under these two constraints, T_{comp} could be minimized by choosing s_1 to be $2V + 1$. However, that is not permissible because of (3.8). Choosing s_1 to be $2V$ adds 1 to T_{comp} in some cases. Therefore, the solution for \underline{s} shown in (3.10) is also optimal in terms of the execution time.

With the knowledge of \underline{s}^T and \mathbf{P}^T , the actual mapping can be performed.

⁸it is sufficient to try combinations of these 8 nodes for p and q in (3.11)

- **Node mapping:**

$$\mathbf{P}^T \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \quad (3.12)$$

- **Arc mapping:**

$$\mathbf{P}^T \mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \left[\begin{array}{c|c|c|c} 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ \hline 1 & 2V+1 & 1 & 0 \end{array} \right] = \begin{bmatrix} 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 0 \end{bmatrix} \quad (3.13)$$

$$\underline{\mathbf{s}}^T \mathbf{D} = [2V \ 0 \ 1] \left[\begin{array}{c|c|c|c} 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ \hline 1 & 2V+1 & 1 & 0 \end{array} \right] = [1 \ | \ 1 \ | \ 1 \ | \ 2V] \quad (3.14)$$

The delays for each pipeline are depicted in (3.14).

- **Input mapping:**

$$\mathbf{P}^T \begin{bmatrix} 3U \\ j \\ k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3U \\ j \\ k \end{bmatrix} = \begin{bmatrix} 3U \\ j \end{bmatrix} \quad (3.15)$$

$$\underline{\mathbf{s}}^T \begin{bmatrix} 3U \\ j \\ k \end{bmatrix} = [2V \ 0 \ 1] \begin{bmatrix} 3U \\ j \\ k \end{bmatrix} = [6UV + k] \quad (3.16)$$

$$k \in \{0, 2V+1, 2 \cdot (2V+1), 3 \cdot (2V+1), \dots\} \quad \text{and} \quad k < K$$

- **Output mapping:**

$$\mathbf{P}^T \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \quad (3.17)$$

$$\underline{\mathbf{s}}^T \begin{bmatrix} i \\ j \\ k \end{bmatrix} = [2V \ 0 \ 1] \begin{bmatrix} i \\ j \\ k \end{bmatrix} = [2Vi + k] \quad (3.18)$$

$$k = 2U \cdot (2V+1) - 1 + m \cdot (2U+1) \cdot (2V+1)$$

$$m \in \left\{ 1, 2, \dots, \left\lceil \frac{I-2U}{2U+1} \right\rceil \right\}$$

The dependence graph is no rectangular parallelepiped, the offset $6UV$ can be subtracted from (3.16) and from (3.18).

As shown in (3.14), all but one pipelines have one delay between two consecutive PEs. The propagation of h requires a large amount of registers ($4UV \cdot (J - 2V)$). Since all PEs along a row share the same h , these values could be broadcasted along each row. Only one register chain with $2V \cdot (2U + 1)$ registers is needed then. If the wire delays involved are too large, the values could be stored at both ends of the array and be broadcasted along a row of PEs toward the center.

The output can be pipelined upward (toward $i = U$). If done, it is available at time $(2V + 1) \cdot i + k$ at the uppermost row of PEs. Then, the output is available at a constant rate (every $2V + 1$ clock cycles). This is the same rate in which the input is needed. The resulting systolic array is shown in figure 3.8. The four pipelines refer to the four arcs that were derived in (3.13). Note that even though the array should consist of J inputs and $J - 2V$ outputs, the rightmost $2V$ columns of PEs do not have to perform any computations. They only propagate data and need no outputs. Each PE consists of a MAC unit, multiplying h and p and adding it to the previous result. It should also be capable of generating a zero, so that a new value of f can be computed. The lowermost row of PEs (see figure 3.10) must have a multiplexer which allows the selection of p from two different sources: either the external input or the value that was stored in each PE to the right. Table 2 shows the listing of a MatLab implementation that simulates such a systolic array and runs the algorithm on it.

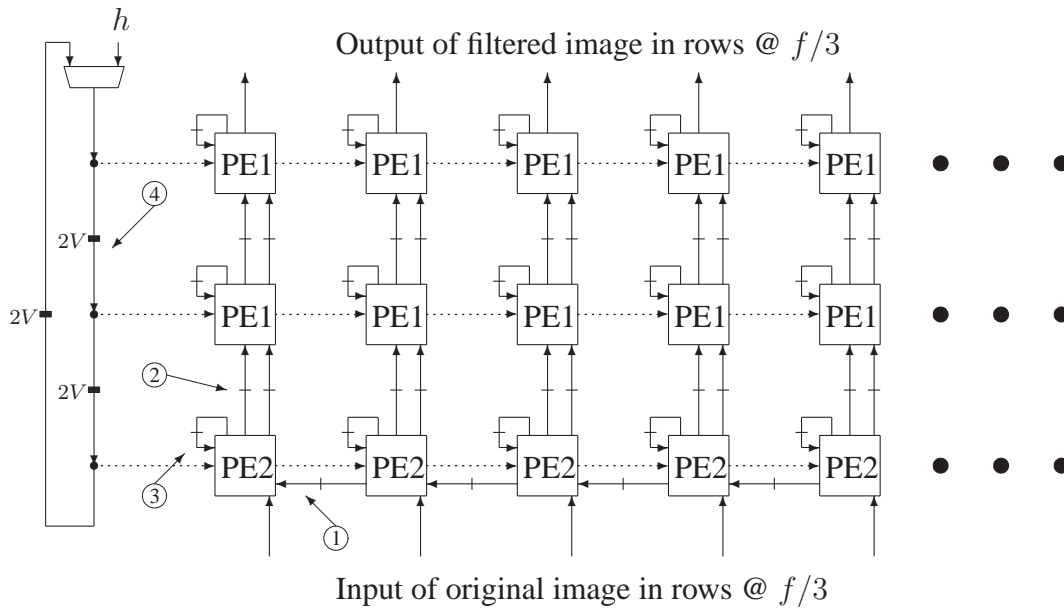


Figure 3.8: Systolic Array 1

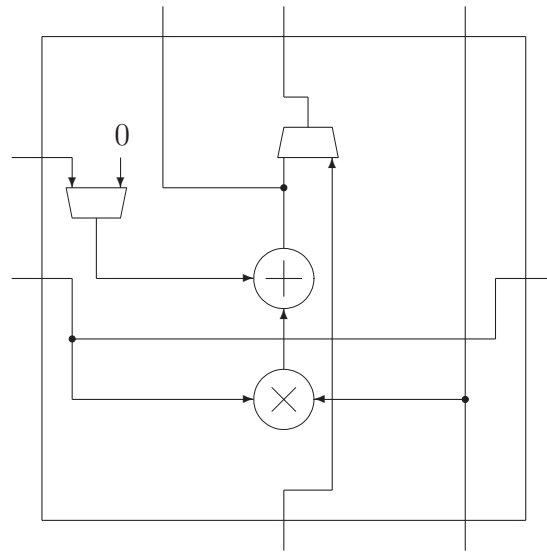


Figure 3.9: PE1

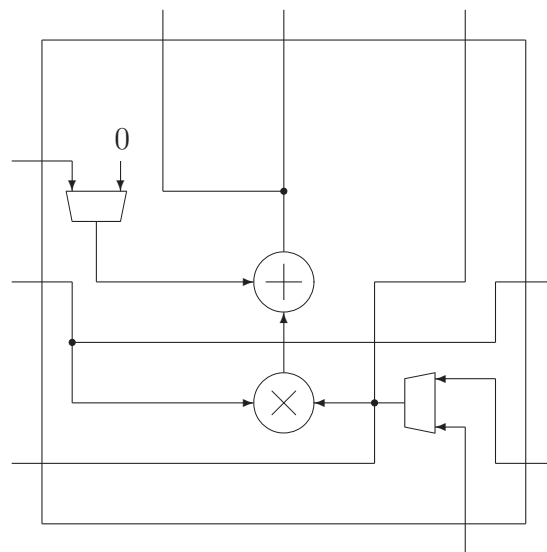


Figure 3.10: PE2

3.3. Multiprojection

The systolic array derived in section 3.2 is customized for a fixed U . If V varies, then the delays along pipeline 4 (see figure 3.8) have to be adjustable. That can be done by a chain of registers and multiplexers which have access to several nodes along that chain. However, if U is not constant, then the array would not be optimal any more. That is because the number of rows of the systolic array is $2U_{max} + 1$, which means that some PEs may not be contributing anything if $U < U_{max}$. Therefore, a second projection along the i direction should be considered. If done, the schedule of the first projection can be the optimal one⁹: $\underline{s}^T = [2V + 1 \ 0 \ 1]$ because $\underline{s}^T v_i \geq 0$ is now permissible.

The resulting systolic array is shown in figure 3.11. Its design is independent from V . Former pipeline 3 now consists of $2U + 1$ registers. Once again, a register chain with $2U_{max} + 1$ registers paired with a multiplexer can ensure that the delay is $2U + 1$ for various U . The input rate is still constant (every $(2U + 1) \cdot (2V + 1)$ clock cycles). The output rate is almost constant: it will be available every $(2U + 1) \cdot (2V + 1) - 1$ clock cycles, where every $(2U + 1)^{th}$ output has an additional delay of $(2U + 1)$, see table 4 on page 27 for a MatLab implementation including these details.

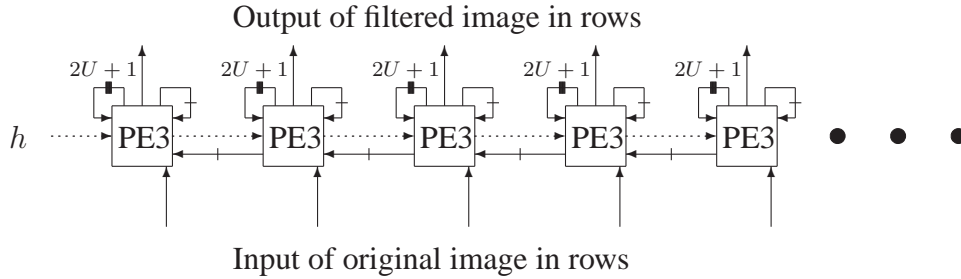


Figure 3.11: Systolic Array 2

3.4. Symmetry

If \mathbf{H} has symmetric entries such that $h(u, v) = h(-u, v)$, then the PEs will compute partial sums of f that can be reused. The lower $U \cdot (2V + 1)$ planes in the dependence graph derived in section 3.1 could be left out. The filtered image would be a sum of the values available at plane $n = (U + 1) \cdot (2V + 1) - 1$ and a shifted intermediate result at plane $n = U \cdot (2V + 1) - 1$. This approach is especially useful for the systolic array derived after multiprojection, as intermediate results can be stored in a cache next to the SA. Dedicated hardware could perform the summation of the two intermediate results to compute f .

For large values of U , up to 50% (in theory) of the computations can be saved. Refer to [5] for further details.

⁹in terms of the total computing time T_{comp}

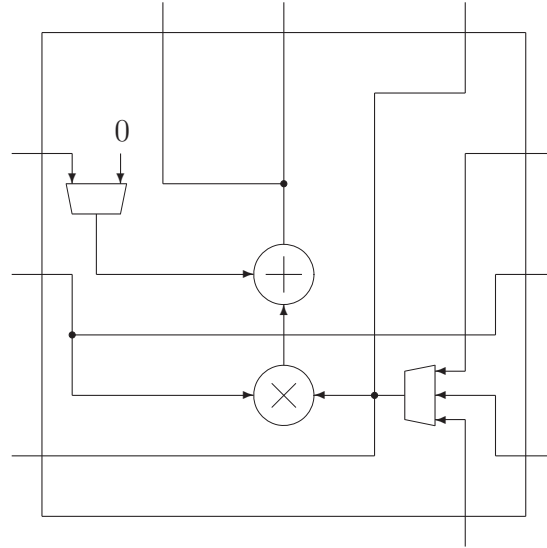


Figure 3.12: PE3

3.5. Partitioning

For "large" images (e.g. $J > 64$), the image has to be partitioned as the number of PEs in one row of the systolic array is limited (chipsize). An LPGS partitioning approach [4] is possible (see figure 3.13). The picture would be divided into I by J_P blocks of pixels. These partitions would be computed sequentially. As some neighboring pixels would appear in two consecutive partitions (see overlap in figure 3.13), some pixels would have to be read from the memory twice. If the SA consists of J_P PEs in each row, $\frac{2V}{J_P}$ would be the overlap. For example, if $J_P = 24$ and $V = 1$, 8.3% of the data would be read twice. This can be avoided by introducing a cache of the size $2VI$ bytes¹⁰. The controlling becomes more difficult in such a case.

¹⁰Assuming that each pixel is represented by 8 bit

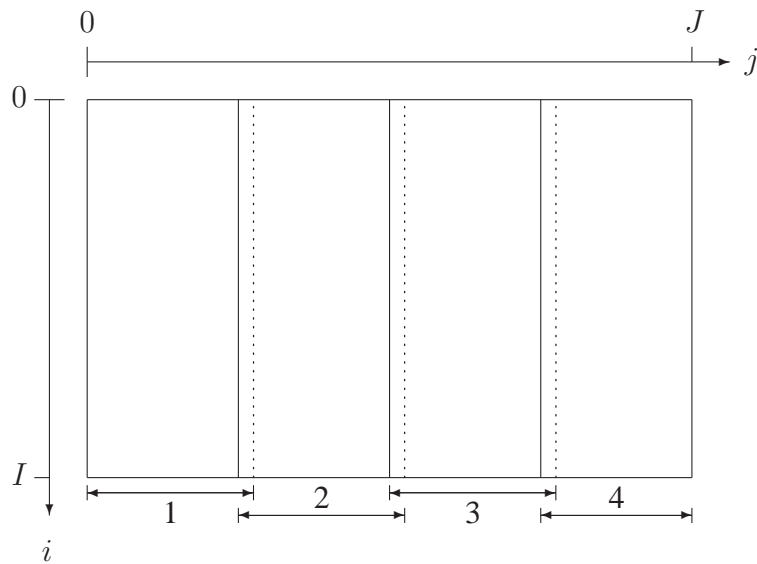


Figure 3.13: Partitioning of the Image, J is divided into 4 partitions

4. Conclusion

2-D filtering is a task that involves a large number of computations. Systolic arrays allow several subtasks to be performed in parallel. The speed up achieved is as follows. The usual sequential execution time is of the order $O[I \cdot J \cdot (2U + 1) \cdot (2V + 1) \cdot T_c]$, where T_c is the cycle time of a basic cell. The architecture suggested in section 3.2 requires significantly less time to perform the task: $O[I \cdot \lceil J/J_P \rceil \cdot (2V + 1) \cdot T_c]$, where J_P is the number of PEs within each row. If the number of PEs matches the one of the architecture suggested by Kung [2] (see section 2.1 for details), both overall computing times are the same. However, the architecture derived in 3.2 is more flexible in terms of how many PEs the array can consist of. Another advantage is its lower I/O rate.

A comparison of the architecture by Hu and King [3] with that derived in 3.2 shows that register usage is more efficient in the solution presented in this report. Also, it allows the SA to consist of multiples of $2U + 1$ PEs, whereas the architecture by Hu and King has a fixed number of PEs: 9.

As symmetry is common for the filter matrices, a main advantage of the architecture derived in 3.3 is the additional speed up that can be gained by exploiting such symmetry. Also, it can easily cope with U and V (representing the filter size) not being constants.

References

- [1] R.C. Gonzalez and R.E. Woods, 2001
Digital Image processing (2nd edition)
Prentice Hall, Upper Saddle Rivwe, New Jersey 07458
- [2] H.T. Kung and S.W. Song, 1981
A systolic 2-D convolution chip
Pittsburgh, Pa.: Carnegie-Mellon University, Dept. of Computer Science,
CMU-CS-81-110.
- [3] Z. Hu and G. King, 1993
A new systolic architecture for 2-D FIR digital filters
General-Purpose Signal-Processing Devices, IEE Colloquium on , 1993 Page(s): 8/1
-8/5
- [4] S.Y. Kung, 1988
VLSI Array Processors
Prentice Hall, Englewood Cliffs, New Jersey 07632.
- [5] H.A. Cohen, 1988
Symmetry Considerations Applied to Hardware Convolvers for Image Filtering
Systems, Man, and Cybernetics, 1988. Proceedings of the 1988 IEEE International
Conference on , Volume: 2 Page(s): 1128 -1131

A. Appendix

```
function f=FIR_2D_1(h,p);
% function f=FIR_2D_1(h,p);
% 2D-FIR filter
% sequential execution
% input:  p ... matrix of image
%        h ... filter matrix
% output: f ... filtered image
% (c) Sebastian Siegel, 05/11/2002

[I,J]=size(p);
[U,V]=size(h);
U=(U-1)/2;
V=(V-1)/2;

f=zeros(I,J);
for i=1+U:I-U,
    for j=1+V:J-V,
%        f(i,j)=0;
        for u=-U:U,
            for v=-V:V,
                f(i,j)=f(i,j)+p(i+u,j+v)*h(u+U+1,v+V+1);
            end
        end
    end
end
end
```

Table 1: 2D-FIR filter version 1

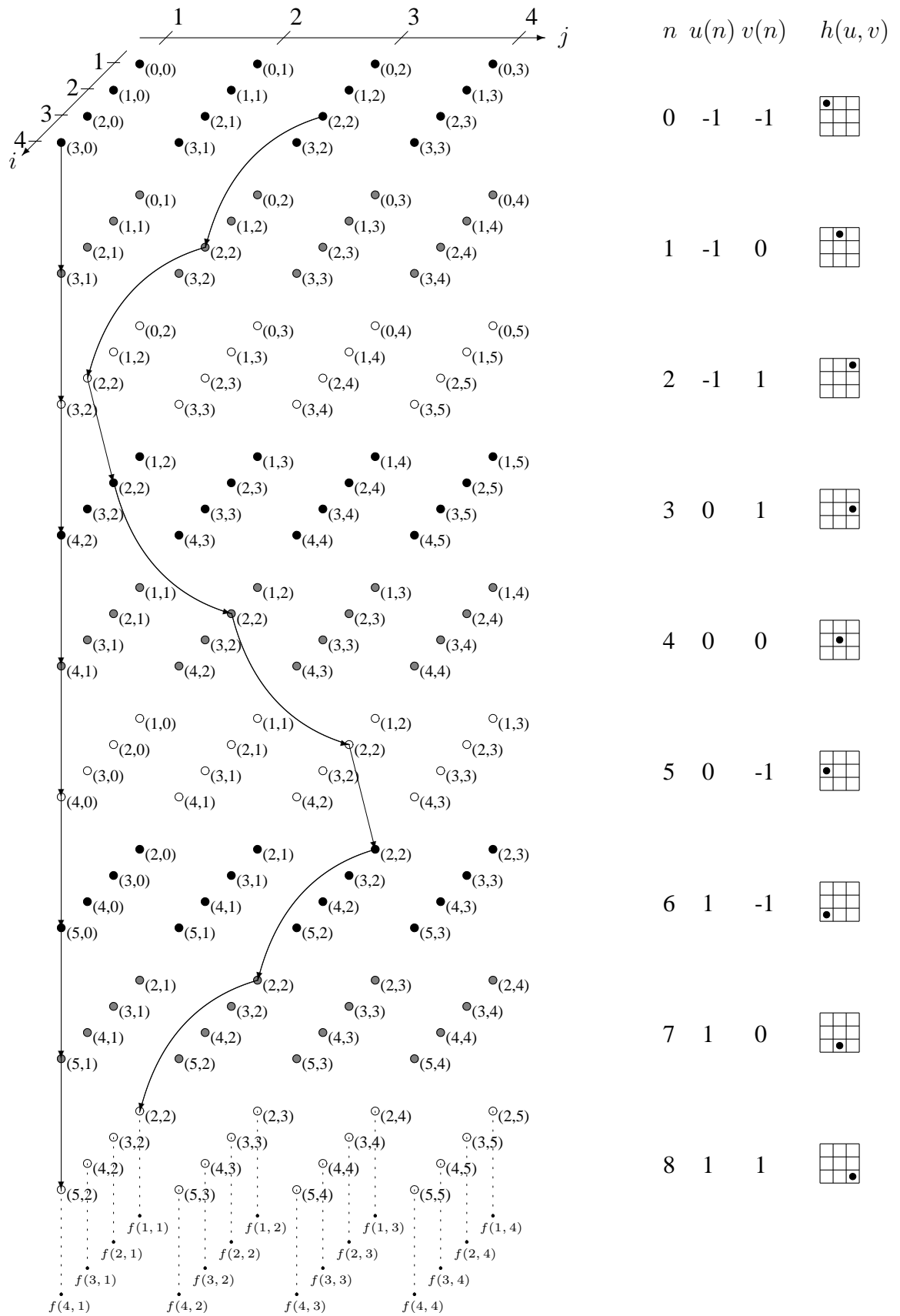


Figure A.1: Dependence Graph ($U = 1$ and $V = 1$). Note that only some dependence vectors are drawn to avoid confusion and that i and j go up to $I - 2$ and $J - 2$ respectively which is not shown in the graph.

```

function f=FIR_2D_2(h,p);
% function f=FIR_2D_2(h,p);
% 2D-FIR filter
% simulated a systolic array, requires 'PEs.m'
% input:  p ... matrix of image
%        h ... filter matrix
% output: f ... filtered image
% (c) Sebastian Siegel, 05/11/2002

[I,J]=size(p);
[U,V]=size(h);
U=(U-1)/2;
V=(V-1)/2;
offs=1;

P1=zeros(1,J);           % pipeline 1 for p
P2=zeros(2*U+1,J);       % pipeline 2 for p
P3=zeros(2*U+1,J-2*V);   % partial sums
P4=zeros((2*U+1)*(2*V),1); % resets P3 if value==1
P5=zeros((2*U+1)*(2*V),1); % filter coeff.
P6=zeros(2*U+1+1,J-2*V); % output pipeline
P7=zeros((2*U+1)*(2*V),1); % copys P3 to P6 if value==1

f=zeros(U,J);

k_reset=2*U;
k_out=2*U-1;
k_max=abs([2*V 0 1]*[U+2*U-(U+mod(I,2*U+1));V-V;...
    0-(2*U*(2*V+1)-1+ceil((I-2*U)/(2*U+1))*(2*U+1)*(2*V+1))])+1;
for k=0:k_max-1+mod(I,2*U+1),
    if mod(k,2*V+1)==0, % read in external data
        readPin=1;
        if k<I*(2*V+1), % still data to read
            Pin=p(k/(2*V+1)+offs,:);
        else
            % last DG-partition, no more data available
            Pin=zeros(1,J);
        end
    else
        readPin=0;
    end
    if mod(k,(2*U+1)*(2*V+1))==k_reset, % generate zeros in PEs
        P4(1)=1;
    else
        P4(1)=0;
    end
    if mod(k,(2*U+1)*(2*V+1))==k_out, % read PEs
        P7(1)=1;
    else
        P7(1)=0;
    end
    u=floor((mod(k-k_reset,(2*U+1)*(2*V+1)))/(2*V+1))-U;
    v=mod(mod(k-k_reset,(2*U+1)*(2*V+1)),2*V+1)-V;
    P5(1)=h(u+U+offs,v+V+offs);
    [P1,P2,P3,P4,P5,P6,P7]=PEs(P1,P2,P3,P4,P5,P6,P7,Pin,readPin,U,V,J);
    if mod(k,(2*V+1))==mod(k_out,2*V+1) & k>=k_out+(2*U+1)*(2*V+1),
        f=[f;[zeros(1,V) P6(1,:) zeros(1,V)]]; % add a new row to output
    end
end
end
f=[f;zeros(U,J)];

```

```

function [P1,P2,P3,P4,P5,P6,P7]=...
    PEs(P1,P2,P3,P4,P5,P6,P7,Pin,readPin,U,V,J);
% function [P1,P2,P3,P4,P5,P6,P7]=...
%     PEs(P1,P2,P3,P4,P5,P6,P7,Pin,readPin,U,V,J);
% 'Systolic Array' needed by 'FIR_2D_2.m'
% input:  P1..P7 ... pipelines
%         Pin    ... input of Systolic Array
%         readPin ... controls Pin
%         U,V    ... filter size
%         J      ... number of columns of PEs
% output: P1..P7 ... pipelines
% (c) Sebastian Siegel, 05/11/2002

if readPin==1, % Pipelines 1&2 get external input
    P2(2*U+1,:)=Pin;
    P1=Pin;
end

for i=1:2*U+1,
    if P4((i-1)*2*V+1)==1,
        P3(i,:)=zeros(1,J-2*V);
    end
end

% actual computation
P3=P3+diag(P5(1:2*V:(2*U+1)*2*V))*P2(:,1:J-2*V);

% copy P3 to output pipeline P6 if necessary
P6(2:2*U+1+1,:)=...
    P6(2:2*U+1+1,:)-diag(P7(1:2*V:(2*U+1)*2*V))*P6(2:2*U+1+1,:)+...
    diag(P7(1:2*V:(2*U+1)*2*V))*P3;

% shifting (finished clockcycle)
P1=[P1(1,2:J), 0];
P2=[P2(2:2*U+1,:);P1];
P4(2:(2*U+1)*2*V)=P4(1:(2*U+1)*2*V-1);
P5(2:(2*U+1)*2*V)=P5(1:(2*U+1)*2*V-1);
P6(1:2*U+1,:)=P6(2:2*U+1+1,:);
P7(2:(2*U+1)*2*V)=P7(1:(2*U+1)*2*V-1);

```

Table 3: Systolic Array in Matlab

```

function f=FIR_2D_3(h,p);
% function f=FIR_2D_3(h,p);
% 2D-FIR filter
% simulated a systolic array (1 row of PEs)
% input:  p ... matrix of image
%        h ... filter matrix
% output: f ... filtered image
% (c) Sebastian Siegel, 05/11/2002
[I,J]=size(p);
[U,V]=size(h);
U=(U-1)/2;
V=(V-1)/2;
offs=1; % offset to avoid index 0 in matrices
P=zeros(2*U+1,J-2*V); % stores partial sums of f
f=zeros(U,J);
k_reset=2*U;
k_out=2*U-1;
k_max=abs([2*V+1 0 1]*[U+2*U-(U+mod(I,2*U+1));V-V;...
    0-(2*U*(2*V+1)-1+ceil((I-2*U)/(2*U+1))*(2*U+1)*(2*V+1))]+1);
for k=0:k_max-1;
    for i=3*U:-1:U,
        if mod(k,2*V+1)==0 & i==3*U, % read in external data
            readPin=1;
            if k<I*(2*V+1), % still data to read
                Pin=p(k/(2*V+1)+offs,:);
            else % last DG-partition, no more data available
                Pin=zeros(1,J);
            end
        else
            readPin=0;
        end
        if mod(k-(i-U)*(2*V+1),(2*U+1)*(2*V+1))==0; % generate zeros in PEs
            Preset=1;
        else
            Preset=0;
        end
        if mod(k-(i-U)*(2*V+1),(2*U+1)*(2*V+1))==(2*U+1)*(2*V+1)-1 &...
            k>=(2*U+1)*(2*V+1)-1;% read PEs
            Pread=1; % one row of filtered image ready
        else
            Pread=0;
        end
        u=mod(3*U-i+floor(mod((k-2*U*(2*V+1)),(2*U+1)*(2*V+1))/(2*V+1)),2*U+1)-U;
        v=mod(mod(k-2*U*(2*V+1),(2*U+1)*(2*V+1)),2*V+1)-V;
        if Preset==1,
            P(1,:)=zeros(1,J-2*V);
        end
        P(1,:)=P(1,:)+Pin(1,1:J-2*V)*h(u+U+offs,v+V+offs); % PE-computations
        P=[P(2:2*U+1,:); P(1,:)]; % shifts pipeline P
        if Pread==1,
            f=[f;[zeros(1,V) P(2*U+1,:) zeros(1,V)]];
        end
        if i==U,
            Pin=[Pin(2:J), 0]; % shift p left
        end
    end
end
end
f=[f;zeros(U,J)];

```

```

function f=FIR_2D_4(h,p);
% function f=FIR_2D_4(h,p);
% 2D-FIR filter
% slow implementaion of a 3 level nested do-loop program in SAF
% input:  p ... matrix of image
%        h ... filter matrix
% output: f ... filtered image
% (c) Sebastian Siegel, 05/11/2002

[I,J]=size(p);
[U,V]=size(h);
U=(U-1)/2;
V=(V-1)/2;
offs=1; % offset to avoid index 0 in matrices

K=2*U*(2*V+1)-1+ceil((I-2*U)/(2*U+1))*(2*U+1)*(2*V+1);
f=zeros(I,J);

for k=0:K,
    u=floor((mod(k-2*U*(2*V+1),(2*U+1)*(2*V+1)))/(2*V+1))-U;
    v=mod(mod(k-2*U*(2*V+1),(2*U+1)*(2*V+1)),2*V+1)-V;
    for i=3*U:-1:U,
        for j=J-1+V:-1:V,
            if i==3*U,
                if mod(k,2*V+1)==0 & k<I*(2*V+1),
                    pp(i,j,k+offs)=p(k/(2*V+1)+offs,j-V+offs);
                elseif j<J-1+V,
                    pp(i,j,k+offs)=pp(i,j+1,k-1+offs);
                else
                    pp(i,j,k+offs)=0;
                end
            end
            if i<3*U & k>=(3*U-i)*(2*V+1),
                pp(i,j,k+offs)=pp(i+1,j,k-(2*V+1)+offs);
            end
            if mod(k,(2*U+1)*(2*V+1))==2*U*(2*V+1),
                ff(i,j,k+offs)=pp(i,j,k+offs)*h(u+U+offs,v+V+offs);
            elseif k>2*U*(2*V+1),
                ff(i,j,k+offs)=ff(i,j,k-1+offs)+...
                    pp(i,j,k+offs)*h(u+U+offs,v+V+offs);
            end
            if mod(k,(2*U+1)*(2*V+1))==2*U*(2*V+1)-1 & ...
                k >= 2*U*(2*V+1)+(2*U+1)*(2*V+1)-1 & j<J-V,
                if k<K | i<U+ceil(mod(I-2*U-0.5,2*U+1)),
                    f(i+(k-(4*U+1)*(2*V+1)+1)/((2*V+1))+offs,j+offs)=...
                        ff(i,j,k+offs);
                end
            end
        end
    end
end
end
end
end

```

Table 5: 2D-FIR filter version 4