8. An Application Example

In this section a 3 by 3 linear filter [GW77] for image-processing will be presented as an application example for the presented data sequencing concept. The example illustrates the utilization of the data sequencing method and demonstrates how memory optimizations may be applied. Only the data sequencer design will be of interest in this application example. A datapath implementation for the Xilinx XC6200 FPGA [Xil96] has been published in [HHG98]. A KressArray implementation of the computational datapath has also been published in [HHH98c].

First the basic filter design will be explained using two alternative specification methods: the XMDS (appendix A), and MoPL (appendix B). After design input, the application of the optimization method presented in figure 7-27 at page 180 will be demonstrated and their effects will be analyzed for each optimization step. The design optimization reduces the number of memory cycles needed for computation. Generally the degree of the achieved optimization is technology independent but the optimization will be performed on the basis of the technological details of a given hardware environment. For this example the hardware environment will be the Map-oriented Machine with Parallel Data Access (MoM-PDA, appendix D). This prototype provides two parallel memory banks, and supports burst accesses and the scan window optimization. While the optimization of the application example is technology independent, its performance will be evaluated, when executed on the CPLD-based data sequencer implementation of the MoM-PDA.

Additionally the address generator needed for the linear filter application will also be mapped to the KressArray-3. Therefore an application specific data sequencer will be generated as explained in section 6.5 at page 121. Two different mappings will be performed considering two KressArray-3 architectures with DPUs of different complexity. For these mappings the required hardware resources will be discussed.

8.1 The Linear Filter Application

This sections describes the general linear filter application.

Definition 8-1: Linear Filter

A linear filter for image-transformation is an operator, which assigns a new value to a pixel depending on the pixel-value and *N* of its neighbor pixels [MW93]. If the calculation of the modified pixel-value is independent from its position, the filter is called homogenous.

The selection of neighbor pixels is done by moving a window over the image. For each window position all covered pixel are used as source data for the filter. The application example of this chapter will be a 3 by 3 linear filter. Therefore the window will cover 9 pixel. Figur e8-1a at page185 shows an example image the 3 by 3 window placed at the upper left position. The new value of the center pixel p_4^{new} is calculated as follows:

$$p_4^{new} = \frac{1}{j} \cdot \sum_{i=0}^{8} p_i \cdot k_i \qquad \textit{(Eq. 8-1)}$$

The new pixel value is stored in a second array, which will contain the filtered picture after all pixel of the source image are processed.

Since the border pixel of the source image do not have *N=8* neighbor pixel, the filtered picture will be smaller. If an image of the size *x* by *y* pixel is processed, the result image is of the size *(x-2)* by *(y-2)* pixel.

Assumed that the filtered image will be stored in array of the size x by y, it is not necessary to write the new pixel value to the location of the original pixel. Figur e8-1b at page 185 illustrates, that the result pixel may also be stored to the p_0 location. Since the value of p_0 is not needed for the following calculations, merging the buffers for source and destination pixel is possible (see figure 8-1c at page 185). The same memory space is used for the source and result image.

For demonstration of the optimization techniques an example image of a size of *x=22* by *y=11* pixel is chosen. The image is mapped as is to the 2-dimensional data memory. Figure 8-2a at page 186 illustrates the mapping. A video scan starts at the top left position and processes the image row by row until the bottom is reached. The scan window (see figure 8-2b at page 186) is designed to read a 3 by 3 area of the data memory. After the reconfigurable datapath applied the filter operation on the data the result is written back to the upper left scan window position and the scan window is moved.

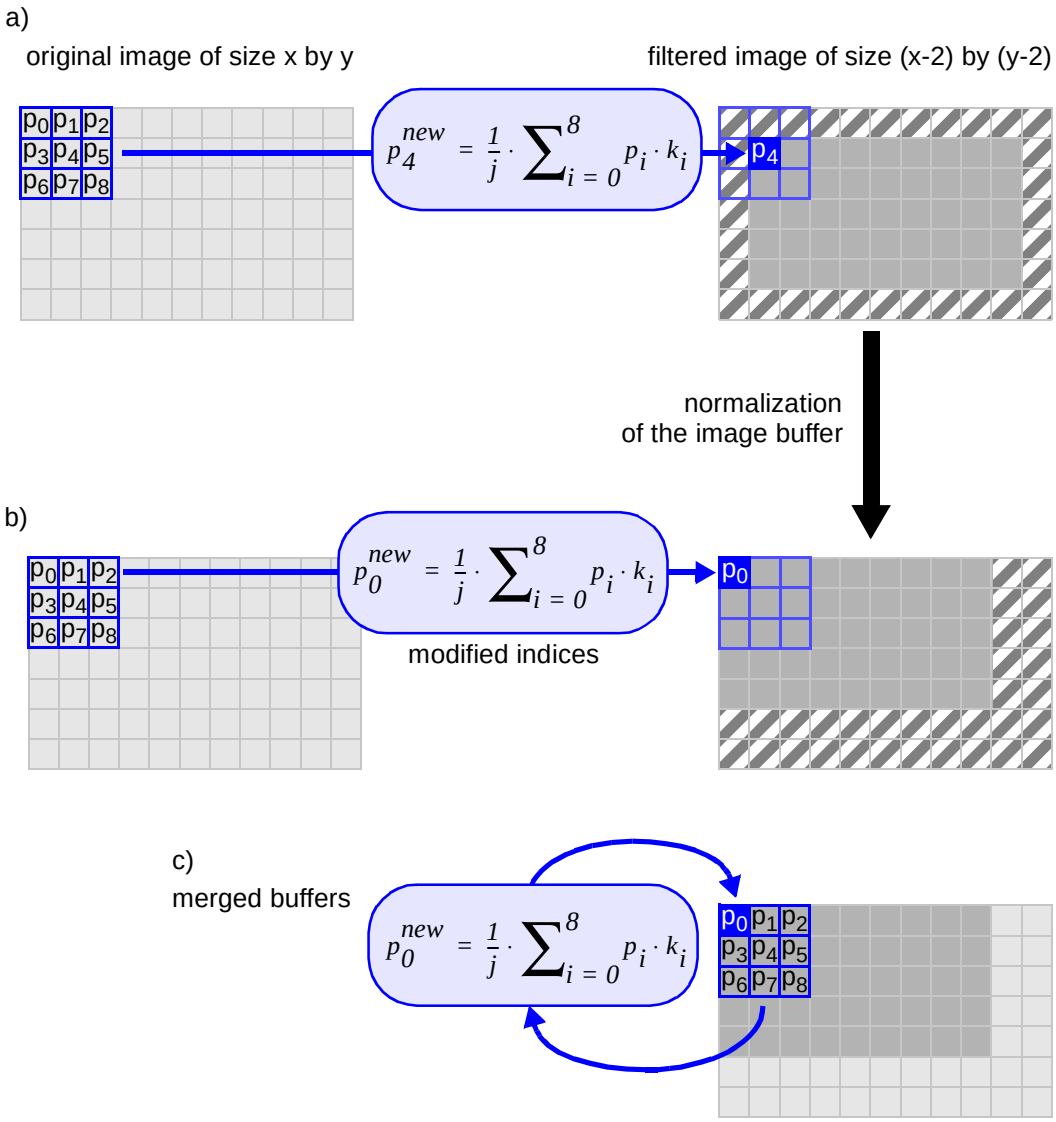


Figure 8-1: The linear filter application:
(a) calculation of a new pixel value under application of a linear filter (two scan windows),
(b) the result may be stored also at the upper left location,
(c) then the result can also be written into the memory space of the original image without overwriting data needed for the following computation steps (one scan window).

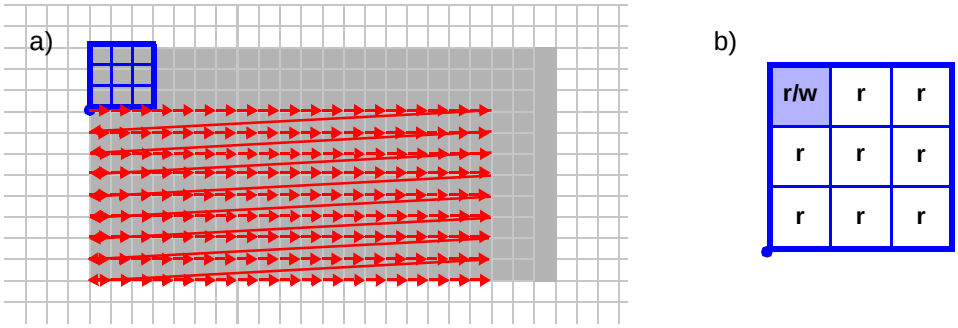


Figure 8-2: Mapping of the merged buffer linear filter implementation to the 2-dimensional memory:
(a) scan pattern, and
(b) initial scan window design.

8.1.1 The Design Specification Using the XMDS

Using the XMDS (see appendix A) a merged buffer design of the linear filter application will be specified with the Task Designer. The Task Designer provides two different user interfaces for scan window input and scan pattern specification.

To enter the required scan window (figure 8-2b), first its x and y size is specified. The lower left area of the scan window input user interface of the Task Designer (see figure 8-3 at page 187) provides two sliders or optional two text fields to enter the scan window size.

The actual scan window is pictured on the right area of the XMDS window (figure 8-3 at page 187). The performed operation of a scan window position is indicated by different colors. The user may modify this by choosing one of the operations (no operation, read, write, or read and write) and selecting the requested scan window position.

After the scan window is entered, the required video scan (figure 8-2) is specified. Figure 8-4 at page 187 illustrates the video scan editor of the XMDS Task Designer. The lower left part of the user interface gives an overview on the actual video scan definition. An arrow shows the starting point of the scan pattern and the direction of the scan line. The same arrow (ΔA) is also illustrated in the right area, where the details of the video scan are specified. The required information includes up to four corner points (*P1*, *P2*, *P3*, and *P4*), the step width (ΔA), and the distance between scan lines (ΔD). Note that these parameters are not the parameters introduced with the slider model (see figure 6-6 at page 100). Further the area specified in the scan pattern input is the area covered by the scan pattern, and not the area covered by the scan window (= image size) while moved by the scan pattern. All parameters for the example application are correctly entered in figur e8-4 at page187.

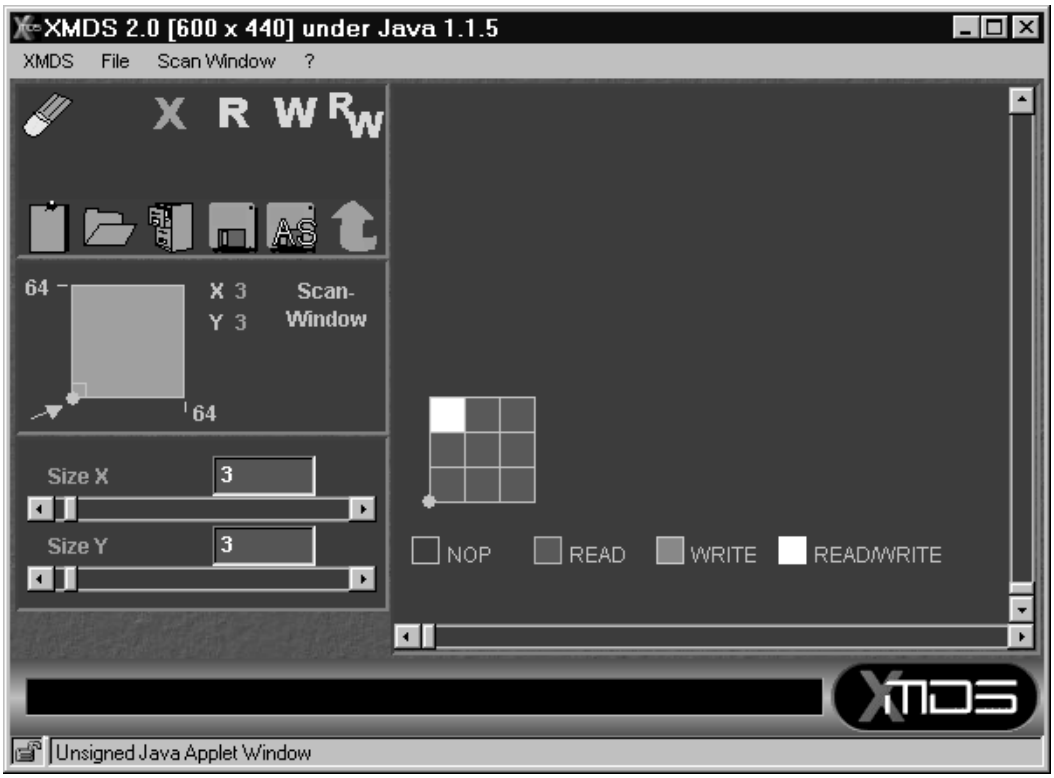


Figure 8-3: Scan window input with the XMDS Task Designer.

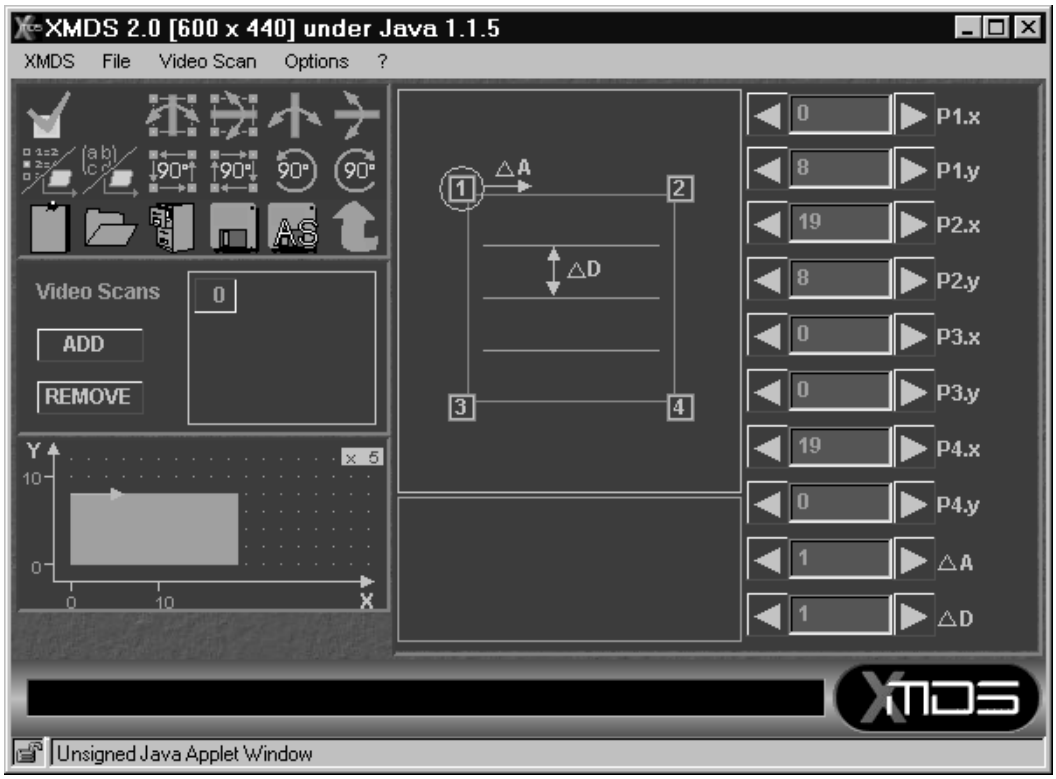


Figure 8-4: Scan pattern input with the XMDS Task Designer.

8.1.2 MoPL Description of the Merged Buffer Linear Filter Application

The following MoPL (see appendi xB) program gives an alternative description of the merged buffer design of the linear filter example. For the example picture *ImgWidth* and *ImgHeight* are defined as follows: *ImgWidth=x=22* and *ImgHeight=y=11*.

```
FUNCTION Filter (ImgWidth, ImgHeight)

/* Set memory area for the image containing integers */
array           ImgData  [1:ImgWidth, 1:ImgHeight] of int;

/* Set scan pattern for RowScan and ColScan according to the algorithm */
scanPattern     RowScan is ImgWidth  - 2 steps [1,0];
                ColScan is ImgHeight - 2 steps [0,1];

/* Set the scan window Img to the size of 3x3 of integers */
window          FilterW is              p [1:3, 1:3] of int;

/* rALU Sharpen sums up the content of every element of Img multiplied by */
/* the corresponding element of Coef to the upper left element of Img */
rALUsubnet      Filter of FilterW is    p [1,3] =   p [1,1] * k0 +
                                                    p [1,2] * k1 +
                                                    p [1,3] * k2 +
                                                    p [2,1] * k3 +
                                                    p [2,2] * k4 +
                                                    p [2,3] * k5 +
                                                    p [3,1] * k6 +
                                                    p [3,2] * k7 +
                                                    p [3,3] * k8;

/* Activation part for the filter algorithm */
begin
/* Specify the scan window group FilterW */
      with FilterW do begin

/* Activate the rALU configuration Filter */
            activate Filter;

/* Move the scan window to its initial positions on ImgData */
            move p to ImgData [1,1];

/* Start the nested scan for Img with RowScan and ColScan */
            RowScan(ColScan[p];)[p];
      end; /* of with FilterW do begin */
end; /* of activation part */
```

8.1.3 Required Memory Cycles of the Merged Buffer Linear Filter Design

The scan pattern for the filter application generates

$$h = (x-2)\cdot(y-2) \qquad \textit{(Eq. 8-2)}$$

handle positions for a *x* by *y* pixel image. For each generated handle position the scan window is positioned and all memory locations determined by the scan window are referenced. The initial scan window reads 9 image pixel and writes back one result pixel (see figure 8-2b at page 186) without any access optimization. Since each memory access requires 2 memory cycles (apply address and access data), equation 8-3 gives the number of memory cycles to access the initial scan window once:

$$ac_{initial} = 2\cdot(9r + 1w) = 20 \qquad \textit{(Eq. 8-3)}$$

The total number of memory cycles for the initial filter application is calculated as follows:

$$app_{initial} = ac_{initial}\cdot h = 20\cdot(x-2)\cdot(y-2) \qquad \textit{(Eq. 8-4)}$$

For the example image of *x=22* by *y=11* pixel the total number of memory cycles without any optimization is:

$$app_{unoptimized} = 20\cdot(22-2)\cdot(11-2) = 3600 \qquad \textit{(Eq. 8-5)}$$

8.2 The Hardware Level Memory Access Optimization of the Merged Buffer Linear Filter Application

In this subsection all hardware level optimizations presented in section 7.1.4 at page 153 are applied to show the resulting acceleration. The hardware level optimizations do not modify the scan pattern or scan window. Only the accesses inside the scan window are accelerated. First each hardware level optimization is demonstrated solely. After this the speed-up of a combination of all hardware level optimizations is shown.

Application of the Parallel Memory Bank Optimization Only

The parallel memory bank optimization may be performed for all handle positions. Having two parallel memory banks, two rows of the scan window are read concurrently. A third row must be read by one bank only, while the other banks stays idle. Also the write access has to be performed by a single bank. Figure 8-5a at

page 190 illustrates the modified scan window. Equation 8-6 calculates the number of required memory cycles. To access the data of the optimized scan window, 14 memory cycles are needed (apply address and access data for 6 read and 1 write operation).

$$ac_{parallel} = 2 \cdot (6r + 1w) = 14 \qquad \textit{(Eq. 8-6)}$$

Application of the Burst Access Optimization Only

The burst access optimization may be performed for all handle positions. All read accesses are done in a burst of length three (see figure 8-5b). For 3 read accesses the burst requires only 4 memory cycles (apply address and 3 times read data). As a result the time to access the data in the modified scan window requires 14 memory cycles (3 bursts and apply address and write data):

$$ac_{burst} = (3 \cdot (1 + 3))r + (1 + 1)w = 14 \qquad \textit{(Eq. 8-7)}$$

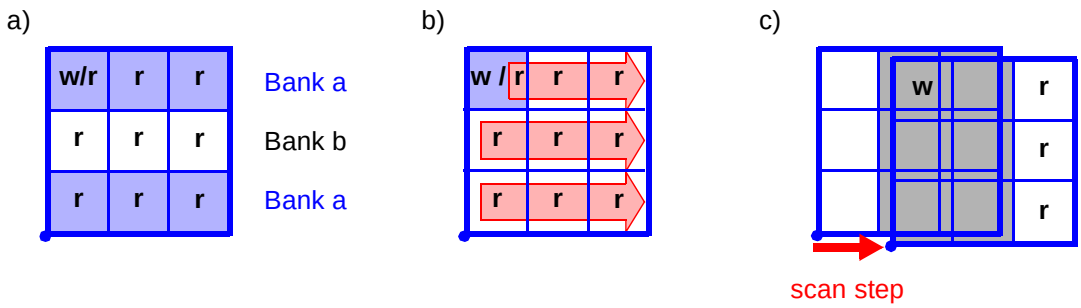


Figure 8-5: Hardware level scan window optimizations for the merged buffer linear filter:
(a) parallel data access only,
(b) burst access only, and
(c) scan window overlap only.

Optimization by Scan Window Overlap Utilization Only

Since the scan window size is lager than the scan step width, also scan window overlapping occurs. The scan window overlap optimization (figure 8-5c) may be used for all handle positions of a scan line except of the first position. Here the scan window is initially placed and no overlap occurs. Since the step width of the scan pattern is 1, two columns of the scan windows overlap during scan window movement. Therefore only the 3 non-overlapping memory locations have to be read and the result has to be

written back. Under application of this modification the inner scan line scan window needs only 8 memory cycles (apply address and access data for 3 read and 1 write operation):

$$ac_{overlap} = 2 \cdot (3r + 1w) = 8 \qquad \textit{(Eq. 8-8)}$$

A Combination of the three Hardware Level Memory Access Optimizations to Achieve the Optimum Acceleration

A combination of the three presented hardware level optimizations will be applied to demonstrate the overall hardware level optimization. Parallel data access may be used for each handle position. Burst access is used for the first handle position of a scan line only. Here no scan window overlap occurs. For inner scan line positions scan window overlap can be exploited and many data accesses are saved completely. Because of this all bursts of inner scan line scan windows would be of the length 1.

In the first scan line position a combination of parallel data access and burst access allows to perform two bursts to the two parallel banks concurrently. Therefore the execution time of the first scan window position of a scan line is 10 memory cycles (two concurrent burst plus one more burst plus, and apply address and write data).

$$ac_{optHL1st} = (2 \cdot (1 + 3))r + (1 + 1)w = 10 \qquad \textit{(Eq. 8-9)}$$

In the inner scan line positions a combination of the parallel data access optimization and scan window overlap optimization is used for the inner scan line scan window. Because of the scan window overlap only the three non-overlapping scan window positions have to be read. Parallel data access allows to perform two read cycles concurrently. Therefore the execution time of the inner scan line scan windows is 6 memory cycles (apply address and access data for two concurrent reads plus one more read plus write result).

$$ac_{optHLinner} = 2 \cdot (2r + 1w) = 6 \qquad \textit{(Eq. 8-10)}$$

Overall Execution Time With Hardware Level Access Optimizations

Figure 8-6 at page 192 compares the number of memory cycles of the different scan window types for the linear filter application. Besides the initial scan window, the hardware level optimizations are illustrated individually, and the final scan window designs are shown.

Equation 8-11 gives the number of memory cycles for the linear filter application optimized on hardware level:

$$app_{HL} = ac_{optHLinner} \cdot (x - 2 - 1) \cdot (y - 2) + ac_{optHL1st} \cdot (y - 2) \qquad \textit{(Eq. 8-11)}$$

For the example image of *x=22* by *y=11* pixel the total number of memory cycles is:

$$app_{HL} = 6 \cdot (22 - 2 - 1) \cdot (11 - 2) + 10 \cdot (11 - 2) = 1116 \qquad \textit{(Eq. 8-12)}$$

Compared to the initial version explained in section 8.1.3 at page 189 with 3600 memory cycles (see equation 8-5 at page 189), the hardware level optimization achieves a speed-up better than three.

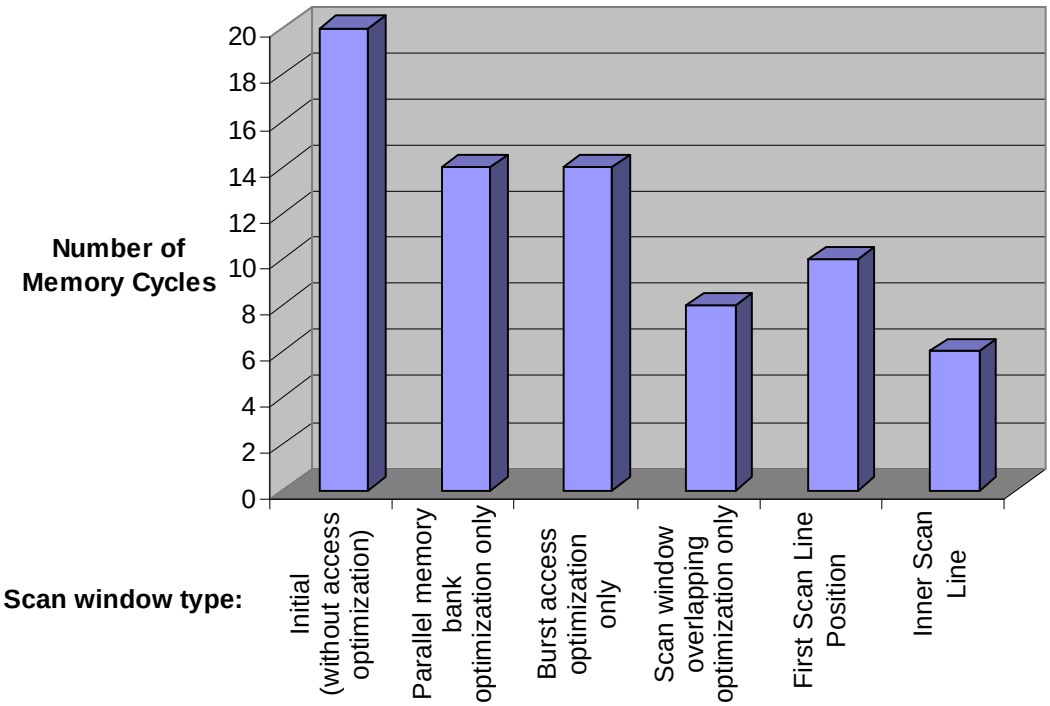


Figure 8-6: Comparison of the different scan window types for the merged buffer linear filter application optimized on hardware level: initial scan window (see section 8.1.3 at page 189), with parallel memory bank optimization only (see page 189), with burst access optimization only (see page 190), with scan window overlapping only (see page 190), and under application of all hardware level access optimizations (see page 191): the first scan line scan window and the inner scan line scan window.

8.3 The Software Level Memory Access Optimization of the Merged Buffer Linear Filter Application

In this section the software level access optimizations method introduced in section 7.5 at page 179 will be applied to the application example. Each step will be carried out individually and its effects will be explained.

Step 1: Data Map Transformation

The data map transformation (see section 7.3.2 at page 168) aims to modify the application, in a way that the scan pattern is parallel to the coordinate axis. Since the initial scan pattern is already parallel to the x- and y- axis (see figure 8-2b at page 186), a transformation of the data map is not needed.

Step 2: Data Map Compression

Data map compression (see section 7.3.2 at page 168) eliminates unused memory locations mixed with the application data. The data of the example application is placed side by side without wasted memory locations in-between (see figure 8-2b at page 186). Therefore no data map compression can be performed.

Step 3: Scan Line Unrolling

In this application example the reconfigurable datapath computation speed and also the question how many operations can be configured into the reconfigurable datapath is not a topic of interest. As a prerequisite it is assumed, that the reconfigurable datapath is capable to process 7 pixel in parallel. In this section it is investigated how loop unrolling (see section 7.2.2 at page 163) improves the memory interface bandwidth.

First it is determined if loop unrolling can be performed. Therefore it is necessary to find an unrolling factor, which divides the number of scan lines. Since the reconfigurable datapath can only process 7 pixel in parallel, the unrolling factor must follow this restriction. At this point of optimization an unrolling factor between 2 and 7 can be chosen. Since the number of scan lines is $(y-2)=9$, $u_1=3$ is the only possible unrolling factor.

Figure 8-7 at page 194 shows the effect of the scan line unrolling (see section7.2.2 at page 163). The number of generated handle positions decreases by the unrolling factor $u_1=3$. It is important to realize that scan line unrolling exploits overlapping scan window positions between different scan lines. The number of overall memory cycles is diminished efficiently. To illustrate this, the number of memory cycles of the application after scan line unrolling is compared with no further optimization to the initial version (see section 8.1.3 at page 189):

After scan line unrolling with the unrolling factor u_1, $h_{SLunrolled}$ handle positions are generated:

$$h_{SLunrolled} = (x-2) \cdot \frac{(y-2)}{u_1} \qquad \textit{(Eq. 8-13)}$$

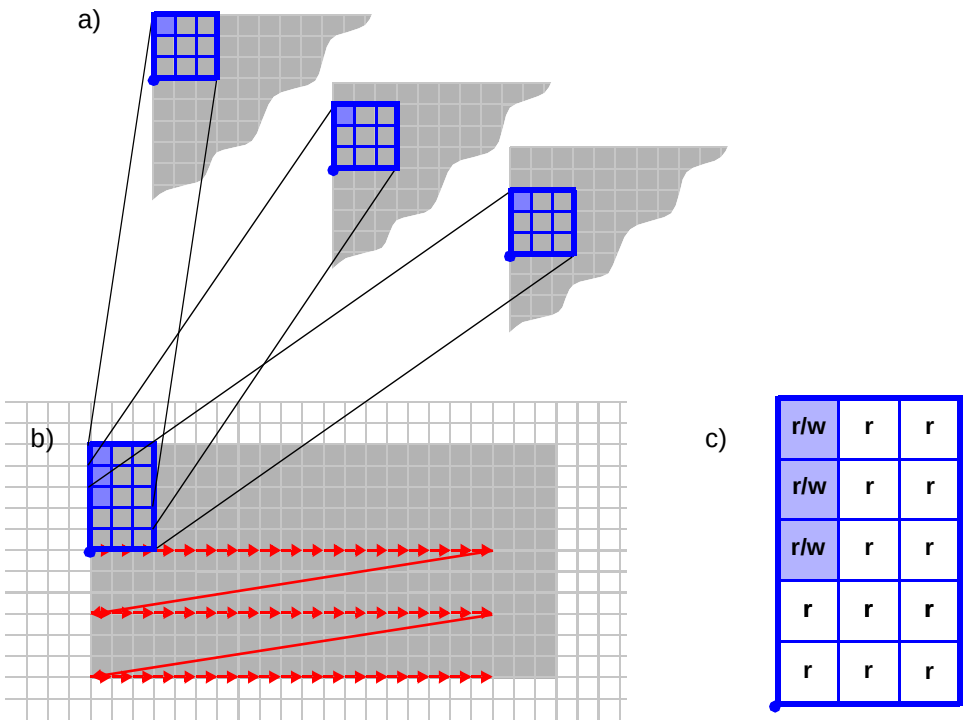


Figure 8-7: Illustration of scan line unrolling:
(a) scan windows at the same position of three subsequent scan lines,
(b) scan line parallelization of the merged buffer linear filter application, generated from the merged buffer linear filter application (see figure 8-2 at page 186) by scan line unrolling, and
(c) its compound scan window.

The number of memory cycles for one handle position produced by the scan window obtained from scan line unrolling without hardware level access optimizations are (see figure 8-7b):

$$ac_{SLunrolled} = 2 \cdot (15r + 3w) = 36 \qquad \textit{(Eq. 8-14)}$$

For each generated handle position the scan window is positioned and all memory locations are referenced. The overall number of memory cycles after scan line unrolling is calculated as follows:

$$app_{SLunrolled} = ac_{SLunrolled} \cdot h_{SLunrolled} \qquad \textit{(Eq. 8-15)}$$

For the example image of $x=22$ by $y=11$ pixel the total number of memory cycles is:

$$app_{SLunrolled} = \frac{36 \cdot (22-2) \cdot (11-2)}{3} = 2160 \qquad \textit{(Eq. 8-16)}$$

The total number of memory cycles is decreased by more than a third of the original value (3600, see equation 8-5 at page 189) through scan line unrolling.

Under application of the hardware level optimizations, scan line unrolling reduces the number of memory cycles by 45 percent (see figure8-12 at pa ge202). The number of memory cycles with hardware level optimizations before scan line unrolling has been determined in equation 8-12 at page 202. The number of memory cycles with hardware level optimizations needed after scan line unrolling is calculated as follows:

The hardware level access optimization techniques distinguish between the first scan window in a scan line and the inner scan line scan windows. The reason for this is, that for the first scan window in a scan line no scan window overlapping occurs.

In the first scan line position a combination of parallel data access and burst access allows to perform bursts to the two parallel banks concurrently. The scan window in figure 8-7c at page 194 performs two read bursts to one memory bank and three read bursts to the other. Since the accesses to the two memory banks are performed concurrently, the three bursts each of length three are taken into account. After the read accesses one and two write accesses are performed in parallel. The execution time of the first scan window position is calculated as follows:

$$ac_{SLunrolled1st} = (3 \cdot (1 + 3))r + (2 \cdot (1 + 1))w = 16 \qquad \textit{(Eq. 8-17)}$$

In the inner scan line positions a combination of the parallel data access, and scan window overlap optimization is used for the inner scan line scan window. It benefits from the scan window overlapping and saves the read accesses to the center and left scan window column. First three and two read accesses are performed to the parallel memory banks and after that two and one write access. Since these accesses are performed concurrently the time for the three read and two write accesses has to be considered. The execution time of the inner scan line scan windows is calculated as follows:

$$ac_{SLunrolledinner} = (3 \cdot (1 + 1))r + (2 \cdot (1 + 1))w = 10 \qquad \textit{(Eq. 8-18)}$$

The scan line unrolled example application optimized on hardware level uses the scan pattern of figur e8-7b at page194. The scan pattern generates 20 handle positions per scan line. Since the example video scan has 3 scan lines, 3 times the memory cycles of equation 8-17 are needed and 27 times the memory cycles of equation 8-18. The total number of memory cycles is calculated as follows.

$$app_{SLunrolledHWopt} = ac_{SLunrolledinner} \cdot (x - 2 - 1) \cdot \frac{(y-2)}{u_1} + ac_{SLunrolled1st} \cdot \frac{(y-2)}{u_1} \qquad \textit{(Eq. 8-19)}$$

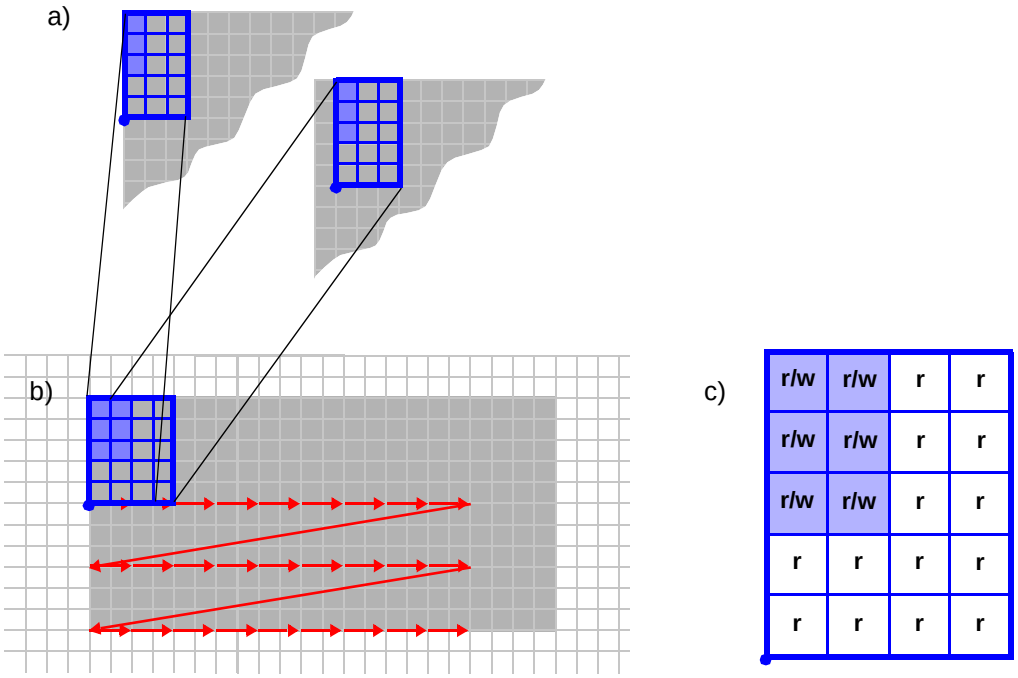


Figure 8-8: Illustration of inner scan line loop unrolling:
(a) scan windows of two subsequent positions in a scan line,
(b) the parallelized merged buffer linear filter application, generated from the step parallelization of the merged buffer linear filter application (see figure 8-7 at page 194) by inner scan line loop unrolling, and
(c) its compound scan window.

For the example image of $x=22$ by $y=11$ pixel the number of memory cycles needed to compute the filter application is:

$$app_{SLunrolledHWopt} = 10 \cdot 19 \cdot 3 + 16 \cdot 3 = 618 \qquad \textit{(Eq. 8-20)}$$

Step 4: Inner Scan Line Loop Unrolling

After scan line unrolling (see section 7.2.1 at page 162) with the unrolling factor $u_1=3$ the capacity of the reconfigurable datapath is not exhausted: 3 pixel are now processed in parallel. Because of the available reconfigurable datapath resources, it would be possible to increase the number of pixel. Since there is no further suitable unrolling factor for scan line unrolling, inner scan line unrolling is considered. The only possible unrolling factor (because of the reconfigurable datapath capacity) is $u_2=2$, which indeed divides the number of scan steps in a scan line (18).

While scan line unrolling reduces the overall number of data accesses, inner scan line unrolling improves only the result of further optimizations. It does not reduce the number of data accesses, because scan window overlap may also be exploited before unrolling (see figure 8-5c at page190).

In this example inner scan line unrolling increases the length of burst accesses, what improves the speed-up of bursts (see equation 7-8 at page 159 and table 7-3 at page 159). Since the number of handle positions is divided by unrolling factor, also the number of burst accesses is divided and the length of the bursts increased. Therefore also number of memory cycles is decreased because the apply address step for half of the initial scan windows is saved for the example application. Under application of the hardware level optimizations, inner scan line loop unrolling reduces the number of memory cycles by 25 percent (see figure 8-12 at page 202). The number of memory cycles with hardware level optimizations before inner scan line loop unrolling has been determined in equation 8-20 at page 212. The number of memory cycles with hardware level optimizations needed after inner scan line loop unrolling is calculated as follows:

The hardware level access optimization techniques distinguish between the first scan window in a scan line and the inner scan line scan windows. The reason for this is, that for the first scan window in a scan line no scan window overlapping occurs.

In the first scan line position a combination of parallel data access and burst access allows to perform bursts to the two parallel banks concurrently. The scan window in figure 8-7c at page 194 performs two read bursts to one memory bank and three read bursts to the other. Since the accesses to the two memory banks are performed concurrently, the three bursts each of length four are taken into account. After the read accesses one and two write bursts each of length two are performed in parallel. The execution time of the first scan window position is calculated as follows:

$$ac_{innnerSLunrolled1st} = (3 \cdot (1 + 4))r + (2 \cdot (1 + 2))w = 21 \qquad \textit{(Eq. 8-21)}$$

In the inner scan line positions a combination of the parallel data access, burst operations, and scan window overlap optimization is used for the inner scan line scan window. It benefits from the scan window overlapping and saves the read accesses to the two left most scan window columns. First three and two read burst accesses of length two are performed to the parallel memory banks and after that two and one write burst accesses of length two. Since these accesses are performed concurrently the time for the three read and two write bursts has to be considered. The execution time of the inner scan line scan windows is calculated as follows:

$$ac_{innnerSLunrolledinner} = (3 \cdot (1 + 2))r + (2 \cdot (1 + 2))w = 15 \qquad \textit{(Eq. 8-22)}$$

The scan line unrolled example application optimized on hardware level uses the scan pattern of figur e8-8b at page196. The scan pattern generates 10 handle positions per scan line. Since the example video scan has 3 scan lines, 3 times the memory cycles of equation 8-17 at page 195 are needed and 27 times the memory cycles of equation 8-18 at page 195. The total number of memory cycles is calculated as follows.

$$app_{innerSLunrolledHWopt} = ac_{innnerSLunrolledinner} \cdot (x-2-1) \cdot \frac{(y-2)}{u_1} + ac_{innnerSLunrolled1st} \cdot \frac{(y-2)}{u_1} \qquad \text{(Eq. 8-23)}$$

For the example image of $x=22$ by $y=11$ pixel the number of memory cycles needed to compute the filter application is:

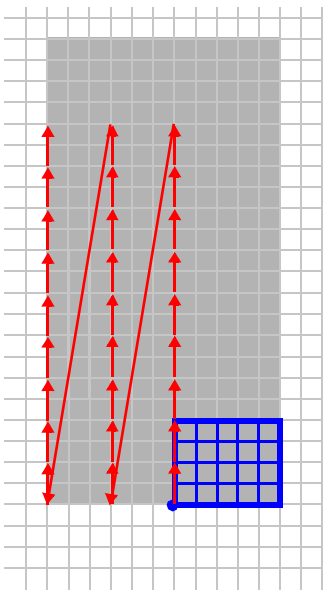
$$app_{innerSLunrolledHWopt} = 15 \cdot 9 \cdot 3 + 21 \cdot 3 = 468 \qquad \text{(Eq. 8-24)}$$

Step 5: Exchange of x And y

In this step (see section 7.3.2 at page 168) it is examined first, if the current scan window (figure 8-8b at page 196) or its mirrored version (figure 8-9b) requires less memory cycles. Since the current scan window has five rows, which results in an unbalanced access to the two parallel memory banks, and the bursts in the mirrored scan window are longer, the application of the exchange x and y step is performed. Figure 8-8 at page 196 shows the resulting scan pattern and scan window.

The modification of the storage map by exchanging x and y exchanges also the effects of burst access and accesses to parallel memory banks. Since parallel memory access of the scan window of figure 8-8b at page 196 is unbalanced this is especially advantageous. The reason for this is, that the previous scan window has 5 rows and

a)



b)

r	r	r	r	r
r	r	r	r	r
r	r	r/w	r/w	r/w
r	r	r/w	r/w	r/w

Figure 8-9: Mapping of the parallelized merged buffer linear filter application:
(a) after modification of the storage map by the exchange of x and y, and
(b) its detailed scan window.

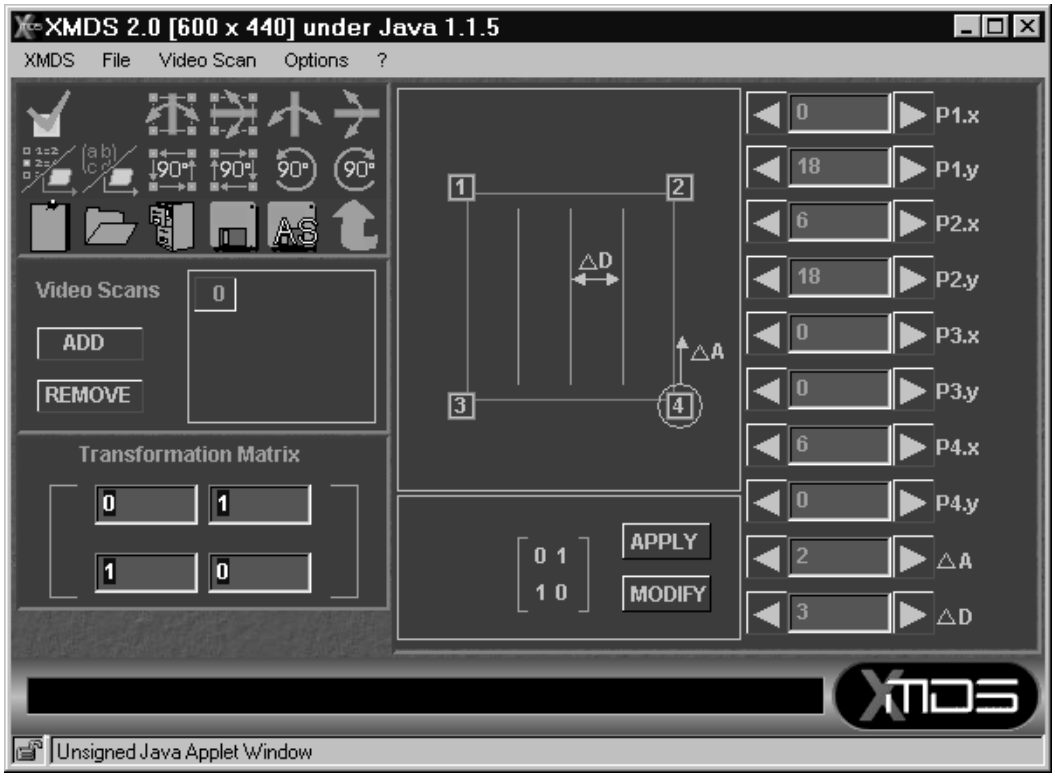


Figure 8-10: Application of the transformation matrix with the XMDS Task Designer.

only 2 parallel banks are available. After the rotation the scan window has 4 rows. Table 7-1 at page 154 documents the speed-up ratio for different numbers of parallel memory banks and the number of accesses.

Additionally the length of the burst operations increases and the number of bursts decreases. This increases also the speed-up of the burst accesses inside the rotated scan window. Furthermore the reduction of the number of burst accesses decreases also the number of configuration words.

The exchange of x and y is performed by the application of the transformation matrix in equation 7-10 at page 169. This may be done using the XMDS Task Designer (see appendix A.2 at page 229). Figure 8-10 illustrates its user interface. Optionally this modification may be programmed by the insertion of rotr and miry and the manual modification of the scan window in the MoPL code (see also section 8.1.2 at page 188):

```
miry(rotr(RowScan(ColScan[Img];)[Img]));
```

The command rotr turns the scan pattern right and the command miry flips y for the scan pattern. For more information on the semantics of MoPL please refer to [ABH94].

Step 6: Low Level Storage Scheme Modification

Since the application calculates a weighted sum (see section 8.1 at page 184) and the addition is a commutative operation, the order in which the sum is built doesn't make any difference. The data may be read in any order in burst mode. Therefore the low level storage scheme is not modified.

Step 7: Scheduling

As explained above the order, in which the data is read or written back, doesn't influence the algorithm speed. Therefore the scheduling considers only the mapping of the 2-dimensional memory to the Multibank DRAM (see figure 7-3 at page 150 and figure 7-10 at page 159). As a result lower scan window rows should be accessed before the higher rows. Since the example image is rather small, the scan window rows may be located in the same row of a memory bank. Further accesses of neighboring rows target different memory banks and are performed concurrently.

Figure 8-11 shows two different scan window designs. Both scan windows perform exactly the same accesses concurrently to the two parallel memory banks. Since scan window overlapping occurs, the scan window of the inner scan line reads only half of the data (figure 8-11b). The scan window of the first handle position of a scan line has to read all the data (figure 8-11a).

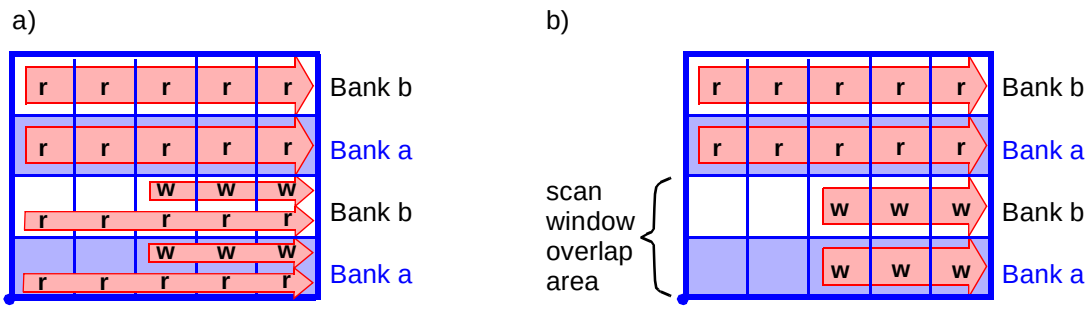


Figure 8-11: Scan window implementation of the parallelized merged buffer linear filter application after modification of the storage map and the access optimization process:
(a) for the first scan line position, and
(b) for inner scan line positions: read bursts of overlapping scan window area omitted.

In the first scan line position (figure 8-11a) a combination of parallel data access and burst access allows to perform bursts to the two parallel banks concurrently. Two burst read accesses each of the length five and one burst write access of the length three are

performed. Therefore it requires 16 memory cycles including all data accesses and apply address cycles. The execution time of the first scan window position is calculated as follows:

$$ac_{opt1st} = (2 \cdot (1 + 5))r + (1 + 3)w = 16 \qquad \textit{(Eq. 8-25)}$$

In the inner scan line positions (figure 8-11b at page 200) a combination of the parallel data access, burst operations and scan window overlap optimization is used for the inner scan line scan window. It benefits from the scan window overlapping and saves one burst read operation. The execution time of the inner scan line scan windows is calculated as follows:

$$ac_{optinner} = (1 + 5)r + (1 + 3)w = 10 \qquad \textit{(Eq. 8-26)}$$

8.4 Memory Access Optimization Results for the Parallelized Merged Buffer Linear Filter Application

The final implementation of the example application uses the scan pattern of figure 8-9a at page 198 and the scan windows of figure 8-11 at page 200. The scan pattern generates 10 handle positions in a scan line. Since the example video scan has 3 scan lines, 3 times the scan window of figure 8-11a at page 200 is applied and 27 times the scan window of figure 8-11b at page 200. Equation 8-27 calculates the number of memory cycles of the fully optimized application. *x* and *y* in equation 8-27 are the size of the initial (*x* and *y* not changed) image.

$$app_{opt} = ac_{optinner} \cdot \left(\frac{x-2}{u_2} - 1\right) \cdot \frac{(y-2)}{u_1} + ac_{opt1st} \cdot \frac{(y-2)}{u_1} \qquad \textit{(Eq. 8-27)}$$

For the example image of *x=22* by *y=11* pixel the final number of memory cycles needed to compute the filter application is:

$$app_{opt} = 10 \cdot \left(\frac{22-2}{2} - 1\right) \cdot \frac{(11-2)}{3} + 16 \cdot \frac{(11-2)}{3} = 318 \qquad \textit{(Eq. 8-28)}$$

Figure 8-12 at page 202 illustrates the memory cycles needed by the filter application with different levels of optimization. While the initial application requires 3600 memory cycles to process the example image of *x=22* by *y=11* pixel, the application optimized with the hardware level optimizations only, requires 1116 memory cycles, after scan line unrolling 618 memory cycles are needed, after inner scan line loop unrolling it requires 468 memory cycles, and the fully optimized implementation needs only 318 memory cycles. This results in the speed-up figures shown in figure 8-13. The

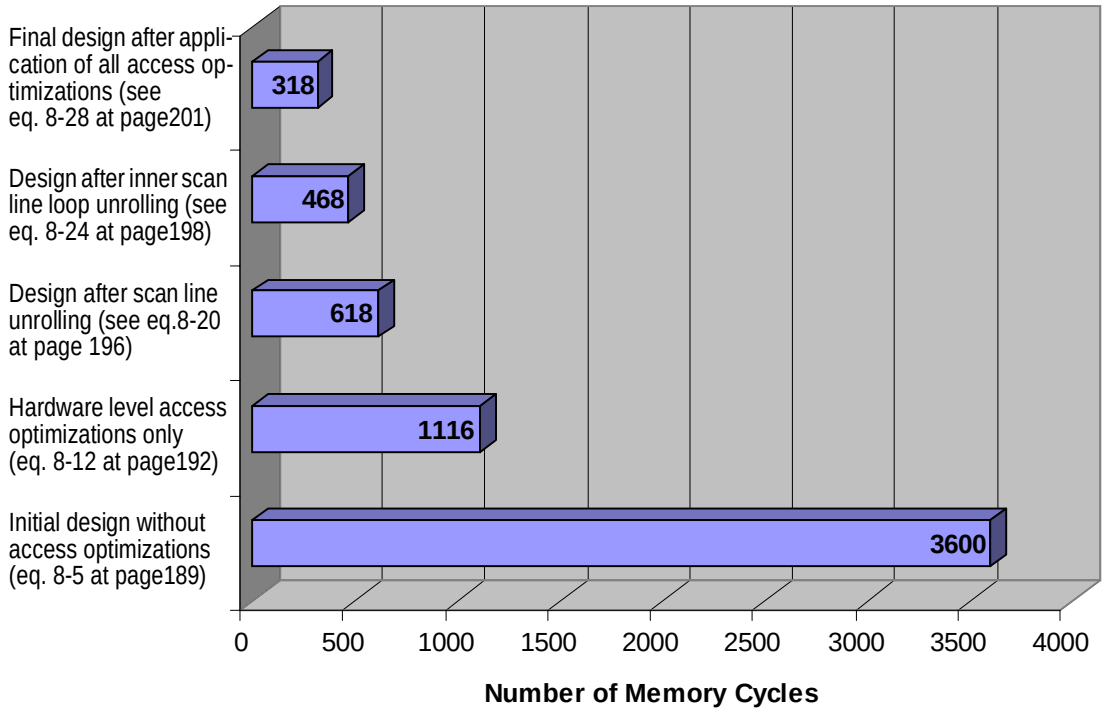


Figure 8-12: Number of memory cycles of the merged buffer linear filter application for different levels of memory access optimization.

hardware level optimizations reduce the number of memory cycles by a factor better than 3. The application of loop transformations, modification of the storage scheme and scheduling of the data accesses improve the performance additionally. The speed-up of the initial application is better than an order of magnitude.

Up to this point all explanations have been technology independent. The presented techniques are targeted to reduce the number of memory cycles generally. The speed-up figures of figure 8-13 consider only the memory interface load. Therefore the time needed by the data sequencer to generate these memory cycles and the practical benefit of the optimizations will be examined in the next section.

8.5 Data Sequencer Performance Evaluation of the Parallelized Merged Buffer Linear Filter Application for the CPLD-based Data Sequencer Solution

The reflection of memory cycles above considers not the time, needed by the data sequencer to generate the memory accesses. Only the memory interface load has been watched. Therefore the initial, un-optimized filter application and the final, fully

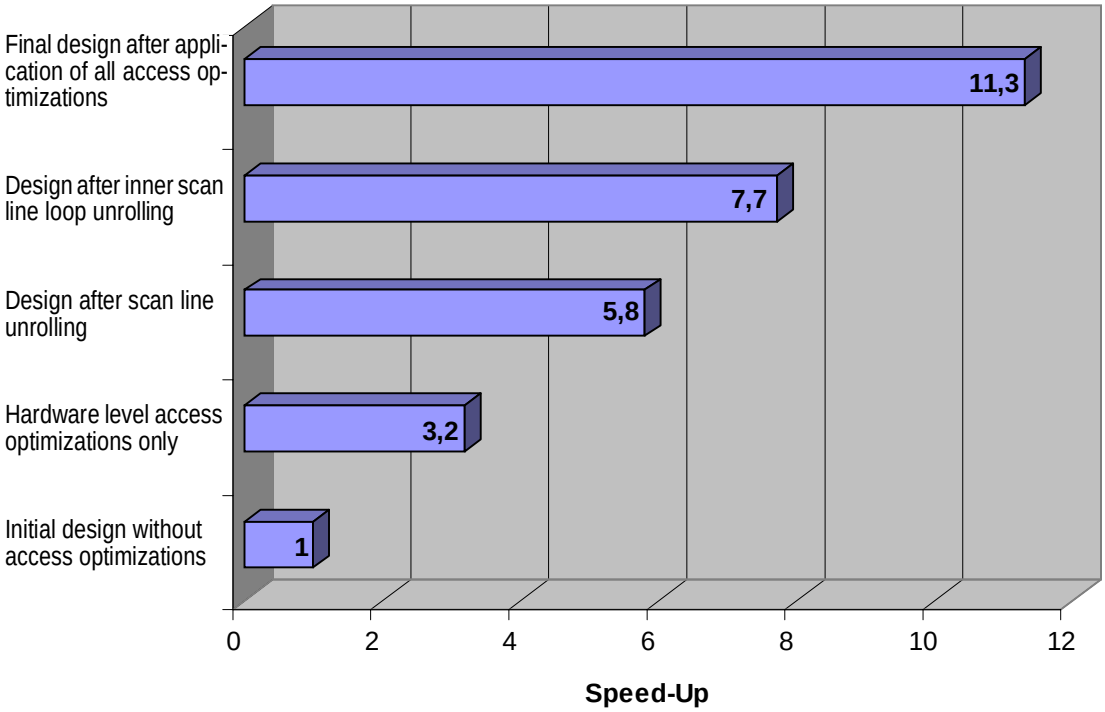


Figure 8-13: Speed-up figures of the merged buffer linear filter application for different levels of memory access optimization. The figures are based on the memory cycles given in figure8- 12 at page202.

optimized version will be examined using the XMDS Application Analyzer (see appendix A.2 at page 229). The basis for the performance evaluation has been the MoM-PDA data sequencer (see appendi xD.2 at page263).

The Performance of the Initial Linear Filter Design

Figure 8-14 at page 204 shows the application statistics report generated by the XMDS Application Analyzer. The execution time of the initial filter implementation (before optimization) with the example image requires 4426 clock cycles. Before execution of the application 183 clock cycles are needed for data sequencer configuration.

Since handle position generation and scan window generation is performed in a pipeline, the Task Statistics report, which pipeline stage is working faster with the current application. Most times (172 vs. 8) the handle position is generated faster with an average difference of 5 clock cycles.

The scan window generation (see Scan Window Statistics in figure 8-14 at page 204) requires 24 clock cycles for each handle position. Each memory access is performed in two clock cycles (apply address and transfer data) and four more clock cycles are needed to load an new handle position.

Application Statistics

Application:	Unnamed, created on the fly
Task count:	1
Clock rate:	50 MHZ
Configuration time:	183 clock cycles = 4 micro seconds
Execution time:	4426 clock cycles = 89 micro seconds

Task Statistics

Task:	Filter_nicht_optimiert
Videoscan count:	1
First videoscan:	0
Execution time:	4426 clock cycles = 89 micro seconds
Scan Window faster:	8 times
Handle Position faster:	172 times
Handle Position equal to Scan Window:	0 times
min Difference:	2 clock cycles = 40 nano seconds ⓘ
median Difference:	5 clock cycles = 100 nano seconds ⓘ
max Difference:	-11 clock cycles = -220 nano seconds ⓘ

Scan Window Statistics

Task:	Filter_nicht_optimiert
Scan Window dimension:	3x3
Number of read commands:	8
Number of write commands:	0
Number of readwrite commands:	1
Used scan window model:	simple
Scan Window execution time:	24 clock cycles = 480 nano seconds

Video Scan 1

Runtime:	3264 clock cycles = 65 micro seconds
SW handles generated:	180
Call count:	180

Figure 8-14: Application Statistics of the initial linear filter implementation generated by the XMDS Application Analyzer.

The Performance of the Final (Optimized) Linear Filter Implementation

Figure 8-15 at page 205 shows the application statistics report generated by the XMDS Application Analyzer. The execution time of the final (optimized) filter application with the example image requires 580 clock cycles. Before execution of the application 102 clock cycles are needed for data sequencer configuration.

Since handle position generation and scan window generation is performed in a pipeline, the Task Statistics report, which pipeline stage is working faster with the current application. Most times (29 vs. 1) the scan window is generated faster with an average difference of 5 clock cycles.

The scan window generation (see Scan Window Statistics in figure 8-15 at page 205) requires 18 clock cycles for the first scan window of a scan line and 14 clock cycles for all other handle positions. All scan window types need one clock cycle per memory

Application Statistics

Application:	Unnamed, created on the fly
Task count:	1
Clock rate:	50 MHZ
Configuration time:	102 clock cycles = 2 micro seconds
Execution time:	580 clock cycles = 12 micro seconds

Task 0 Statistics

Task:	Filter_optimiert
Videoscan count:	1
First videoscan:	0
Execution time:	580 clock cycles = 12 micro seconds
Scan Window faster:	29 times
Handle Position faster:	1 times
Handle Position equal to Scan Window:	0 times
min Difference:	-3 clock cycles = -60 nano seconds ⓘ
median Difference:	-5 clock cycles = -100 nano seconds ⓘ
max Difference:	-22 clock cycles = -440 nano seconds ⓘ

Scan Window Statistics

Task:	Filter_optimiert
Scan Window dimension:	1x1
Used scan window model:	realistic
Number of memory banks:	2
Complete execution time:	432 clock cycles = 9 micro seconds
Execution time of the **first** scan window in a scanline	
Number of read commands:	20
Number of write commands:	6
Call count:	3
Execution time:	54
Memory bank 0:	18 clock cycles = 360 nano seconds
Memory bank 1:	18 clock cycles = 360 nano seconds
Execution time of a **middle** scan window in a scanline	
Number of read commands:	10
Number of write commands:	6
Call count:	24
Execution time:	336
Memory bank 0:	14 clock cycles = 280 nano seconds
Memory bank 1:	14 clock cycles = 280 nano seconds
Execution time of the **last** scan window in a scanline	
Number of read commands:	10
Number of write commands:	6
Call count:	3
Execution time:	42
Memory bank 0:	14 clock cycles = 280 nano seconds
Memory bank 1:	14 clock cycles = 280 nano seconds

Video Scan 1

Runtime:	576 clock cycles = 12 micro seconds
SW handles generated:	30
Call count:	30

Figure 8-15: Application Statistics of the final (optimized) linear filter design generated by the XMDS Application Analyzer.

cycle as described in Step 7 in section 8.3 at page 192. The first scan window of a scan line needs two more clock cycles and all others four additional clock cycles to load an new handle position.

Discussion of the Performance Analysis Results

Figure 8-16 compares the clock cycles of both implementations. The speed-up of the optimized implementation is calculated as follows:

$$speedup_{MoM-PDA} = \frac{4426}{580} = 7,6 \qquad \textit{(Eq. 8-29)}$$

The speed-up considering only the memory cycles has decreased from 11,3 to 7,6 for the MoM-PDA data sequencer implementation. The reason for this is, that during the execution of the initial filter design the pipeline stage, which generates the scan window, dominates the execution time. For the optimized filter implementation the

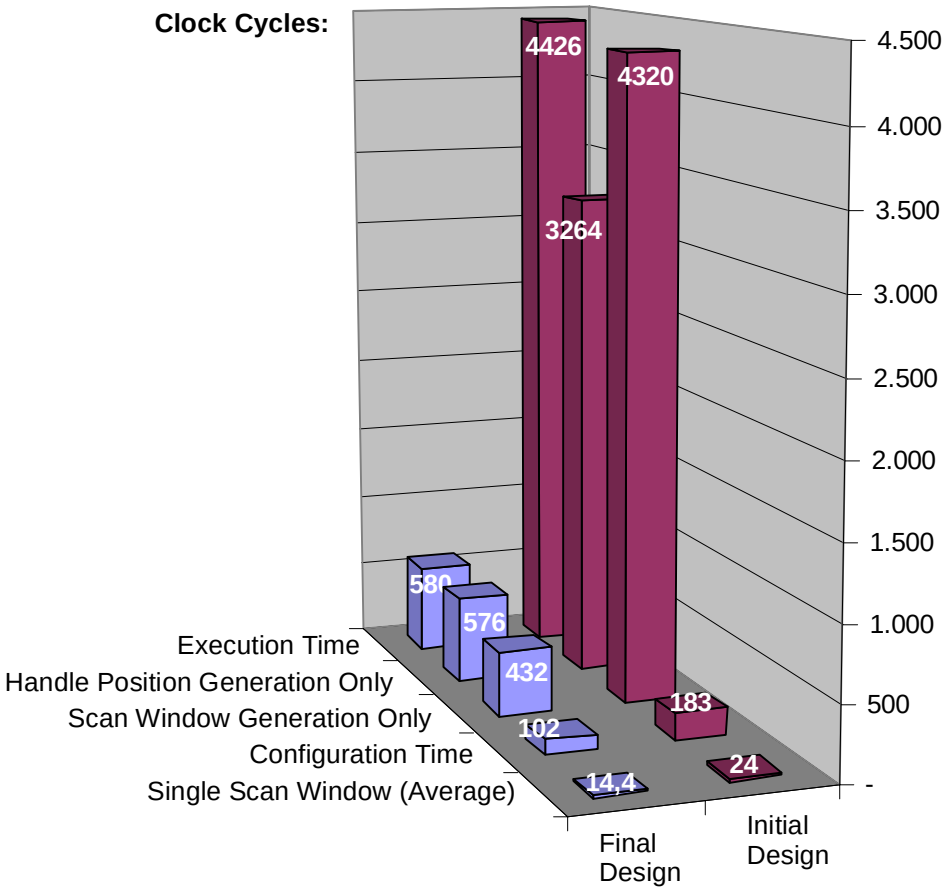


Figure 8-16: Required clock cycles of the initial and final (optimized) filter implementation.

generation of the handle position works slower than the scan window generation. Therefore the second pipeline stage stays idle for an average of 5 clock cycles per handle position.

The main reason for the slow handle position generation of the MoM-PDA data sequencer is the utilized target device. The CPLD-based data sequencer implementation (see appendix D.3.2.1 at page 284) was limited by the Logic Element (LE) count of the CPLD. Therefore the data sequencer lost performance to fit into the device. With a bigger CPLD the handle position generator could have more pipeline stages and would be faster.

With a faster handle position generation, the execution time could be lowered to be the scan window execution time, which is 432 clock cycles (see the Scan Window Statistics in figure 8-15 at page 205). In that case the speed-up would come close to the theoretical value of 11.3:

$$speedup_{optimizedMoM-PDA} = \frac{4426}{432} = 10,2 \qquad \text{(Eq. 8-30)}$$

The Configuration Time

The utilization of burst accesses (section 7.1.4.3 at page 158) causes a remarkable reduction of the configuration time:

$$speedup_{ConfigurationTime} = \frac{183}{102} = 1,8 \qquad \text{(Eq. 8-31)}$$

This effect has been intensified by the loop unrolling technique (section 7.2 at page 162) and the exchange of x and y modification (section 7.3.2 at page 168), because these transformations lengthened the burst accesses.

8.6 A Parallelized Linear Filter Implementation using the KressArray-3

In this subsection the linear filter application presented in section 8.1 at page 184 is used to illustrate how application specific data sequencers may be mapped together with the computational datapath to the same reconfigurable device. A KressArray of $7 \cdot 8 = 56$ DPUs is used for the implementation. The operator array is surrounded by memory units. To exploit the large number of parallel memory banks, different memories are used for source and destination image as shown in figure 8-1b at page 185. As a result data can be read and written at the same time.

To fill the complete array of 36 DPUs, the initial filter design has been parallelized by the scan line unrolling optimization (section 7.2.2 at page 163) with the unrolling factor $u_k=2$. figure 8-17a shows an example input image of the size $x=22$ and $y=10$ with the appropriate video scan. The scan window is illustrated in figure 8-17b. In the output image of the size $x=20$ and $y=8$ with the appropriate video scan is shown. The scan window is illustrated in figure 8-17d.

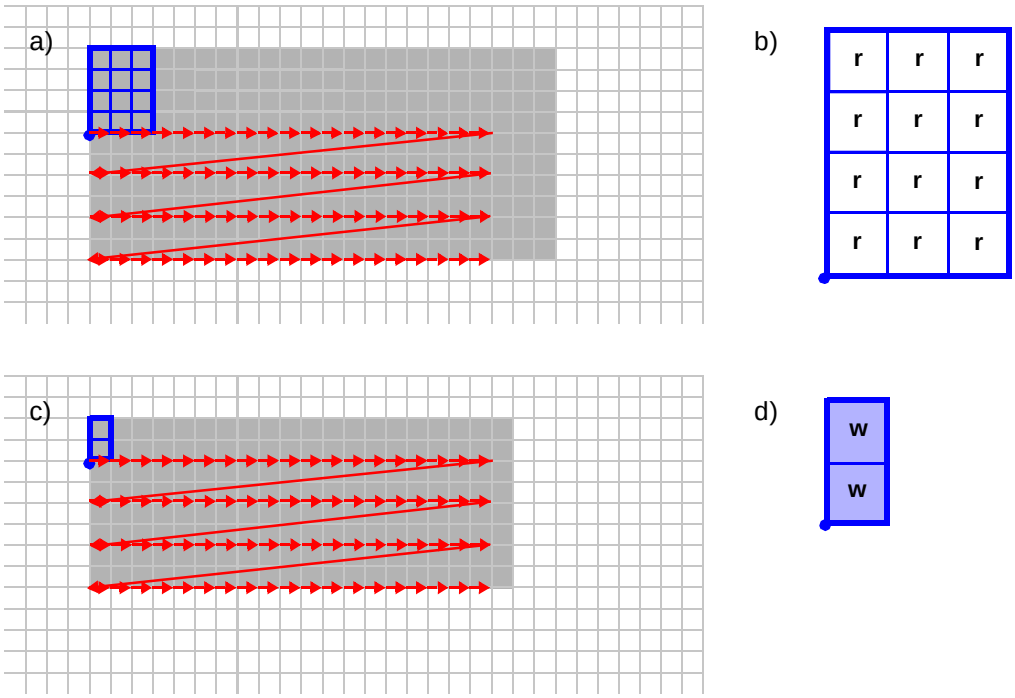


Figure 8-17: The parallelized linear filter implementation used for the KressArray
(a) input image
(b) with scan window, and
(c) filtered result picture
(d) with scan window.

The linear filter implementation will be demonstrated for two different KressArray architectures. The first implementation (section 8.6.1 at page 209) uses the operators of the same granularity as used in section 6.5 at page 121. With the availability of even more coarse operators or operators that may operate sequentially like in the CHESS array (see section 2.9 at page 19), the data sequencer may be implemented with less DPUs. This is demonstrated with a second implementation (section 8.6.2 at page 215).

8.6.1 Example Implementation With Datapath Units of Conventional Functionality

This example implementation utilizes the KressArray surrounded by data memory only, which is introduced in section 6.5.1 at page 123. The granularity of the DPUs is chosen, that each operator may perform one arithmetic operation at a time. An overview on the overall design is given in figure 8-18 at page 210. It consists of the following components:

- an application specific handle position generator for rectangular video scan as pictured in figure 6-29 at page 137,
- four scan window generators, separated for read and write accesses and one for each parallel bank,
- two memory bank switchers (stream mapper operators), which perform the same operation as the memory bank switcher of the hardwired data sequencer implementation (see appendix D.2.2 at page 269),
- four memory banks, two for the source data, and two for result data,
- the reconfigurable ALU Port using data distributor operators (DD), and a stream mapper operator with control input (cSM).
- the computational datapath.

The Stream Mapper Operator

The stream mapper operator is programmed by a constant and has two inputs and three outputs. The programmed constant is the position of the LSB of the y-part of the address. This information is needed because the HPG performs the address mapping described in appendix D.2.2 at page 269. According to the LSB of the y-address part incoming addresses are mapped to one of the two parallel memory banks (see also figure 7-4 at page 151). Additionally the LSB of the y-part of the address is eliminated by an one bit shift of the y-part of the address. The third output is control output to be used to control stream multiplexers with control input.

The Stream Multiplexer Operator With Control Input

The stream multiplexer operator with control input has three inputs and two outputs. The control input is used to determine which input stream is connected to which output port. The two inputs are assigned to the two outputs according to this control signal generated by a stream mapper operator.

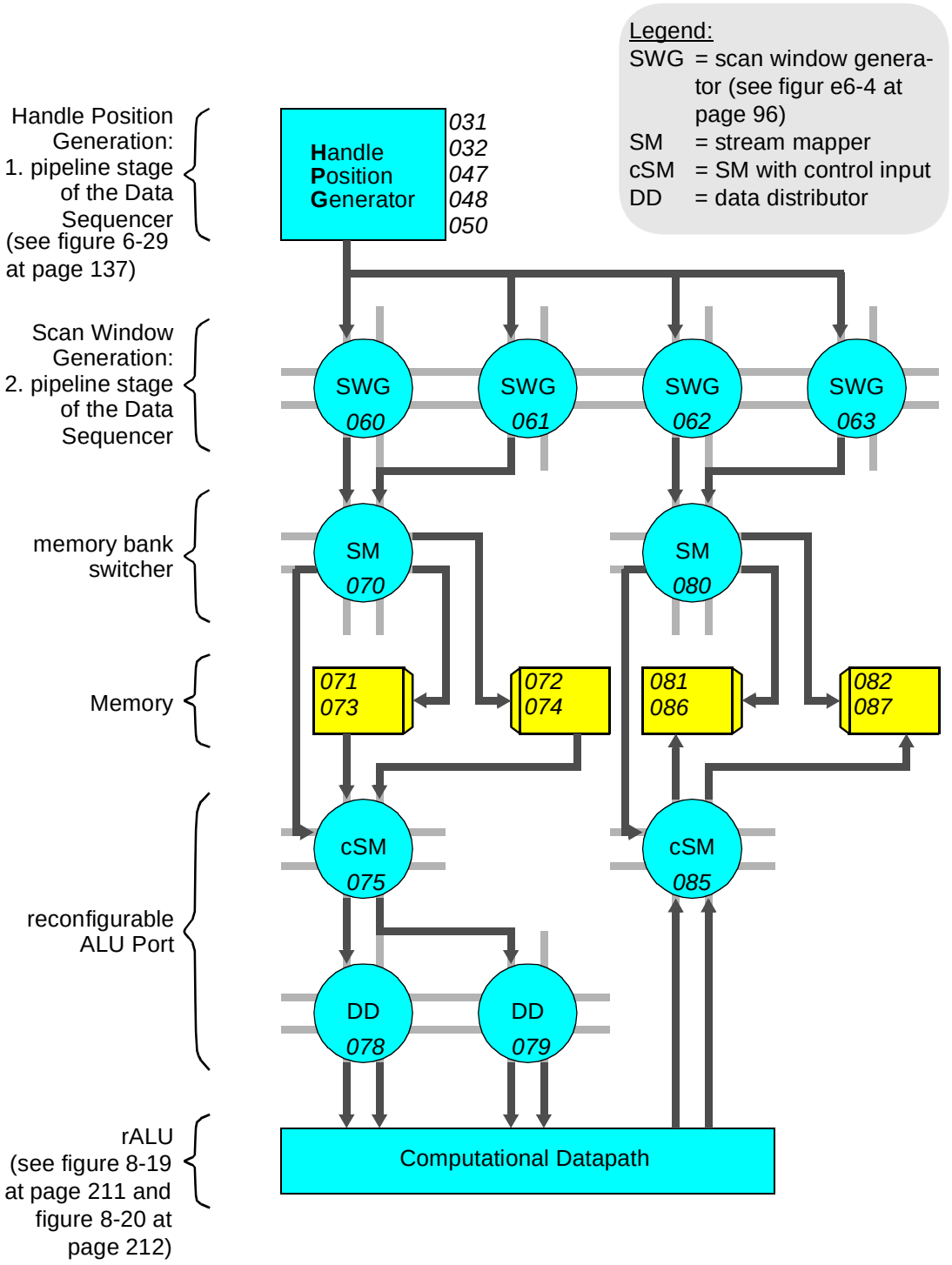


Figure 8-18: Linear filter implementation for the KressArray-3 using DPUs of conventional functionality. Cursive numbers indicate the operator and memory port numbers in the layout, pictured in figure 8-21 at page 214.

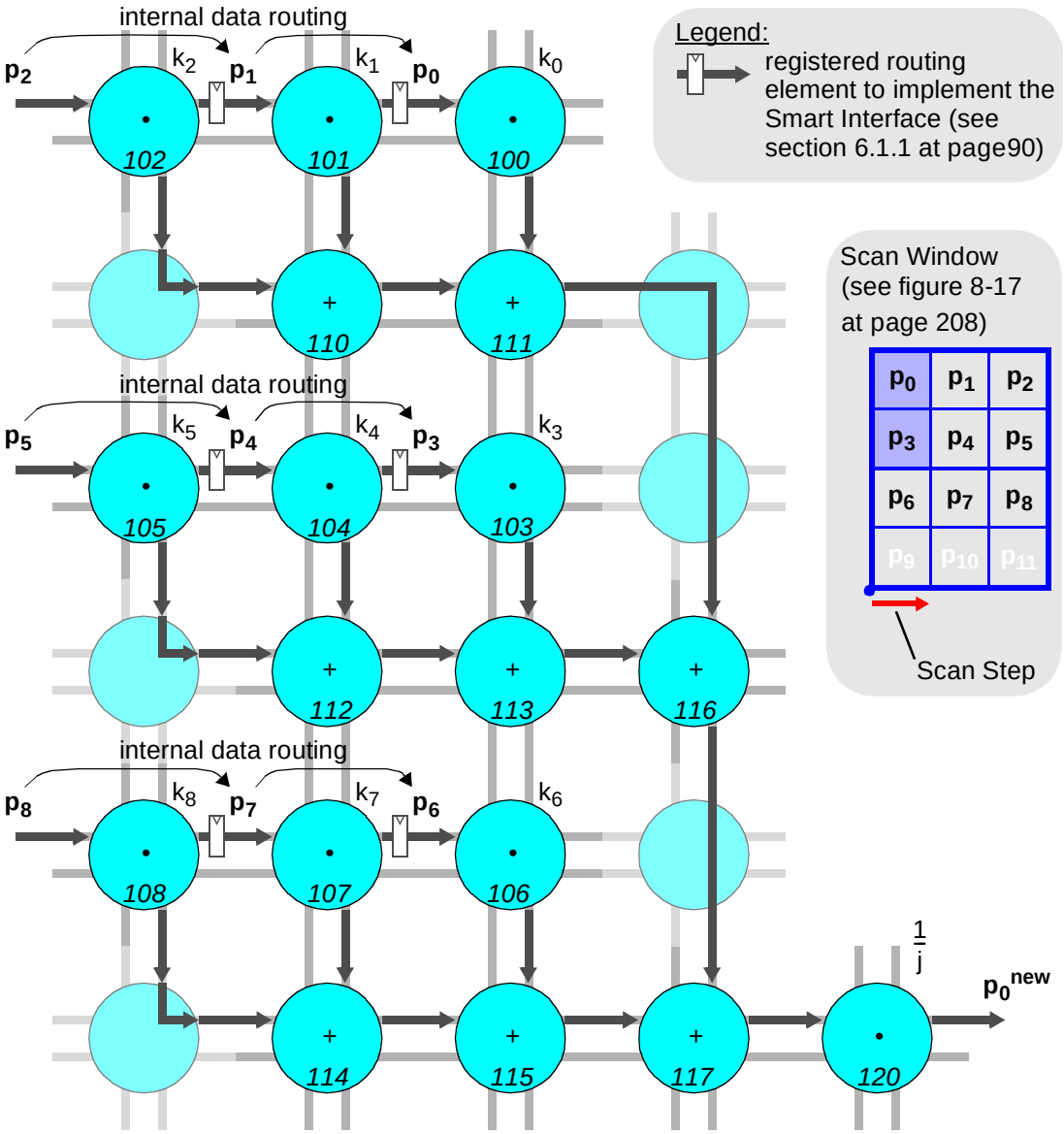


Figure 8-19: Linear filter implementation to compute p_0^{new}. Cursive numbers indicate the operator numbers in the layouts, pictured in figure 8-21 at page 214 and figure 8-23 at page 216.

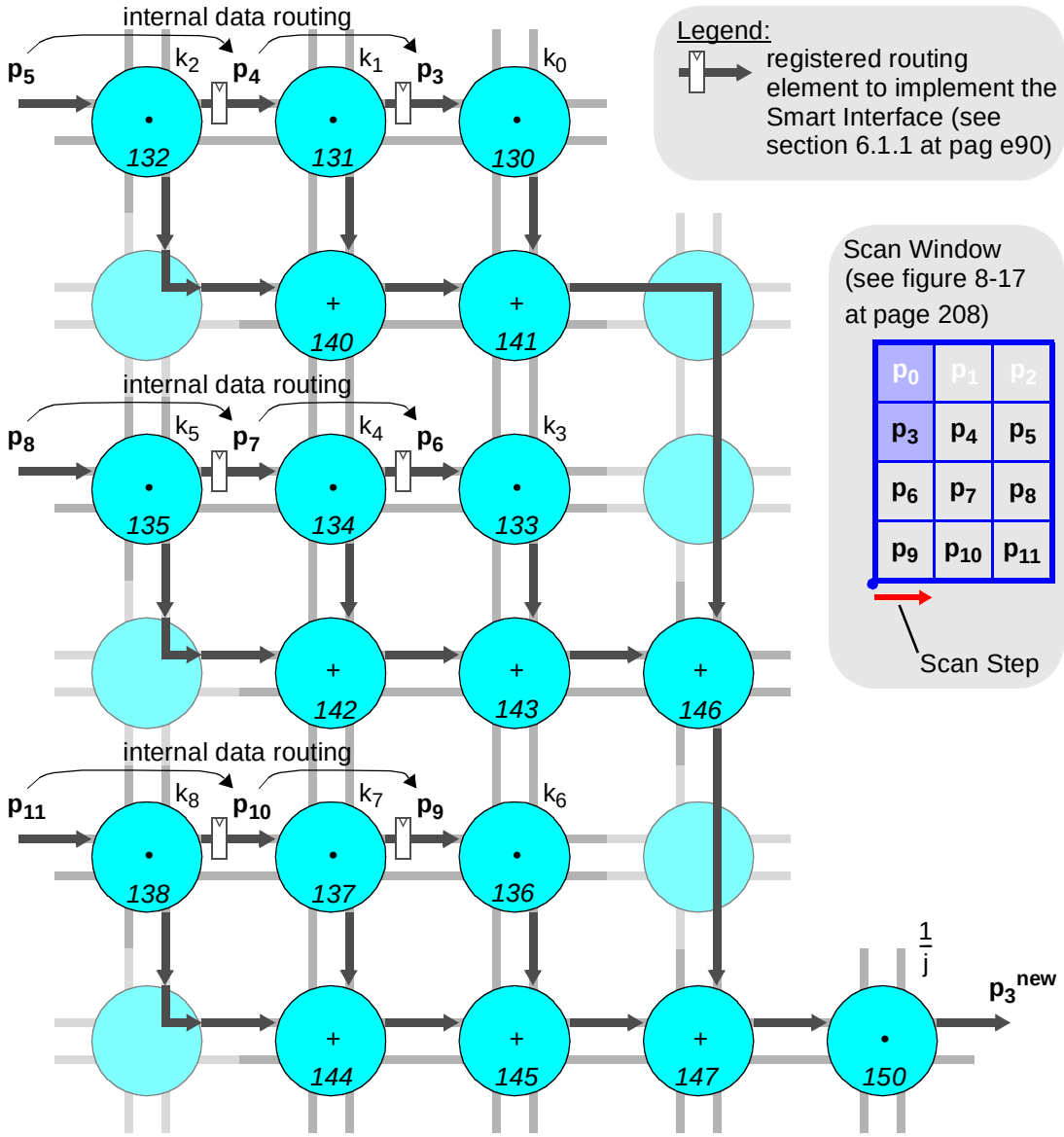


Figure 8-20: Linear filter implementation to compute p_3^{new}. Cursive numbers indicate the operator numbers in the layouts, pictured in figure 8-21 at page 214 and figure 8-23 at page 216.

The Data Distributor Operator

The data distributor operator is programmed by one constant, has one input and two outputs. It distributes the data on the input to the two outputs by the programmed scheduling.

The Datapath Implementation

The computational datapath is programmed to process two pixel in parallel to benefit from the parallel data streams. The datapath for the two pixel is the same. It is shown in figure 8-19 at page 211 and figure 8-20 at page 212. Both share parts of the input data as both need partially the same data to compute a result. Further registered routing operators have been inserted to implement the Smart Register File, which stores the data from the overlapping scan window positions. Each overlapping scan window position has a corresponding registered routing operator. This datapath implementation has also been published in [HHH98c],

The KressArray Mapping

The complete design has been mapped to the KressArray using the MA-DPSS (see [HHH99a] or [HHH00]). Figure 8-21 at page 214 shows the mapping result. 14 of 50 DPUs where used for the data sequencer implementation.

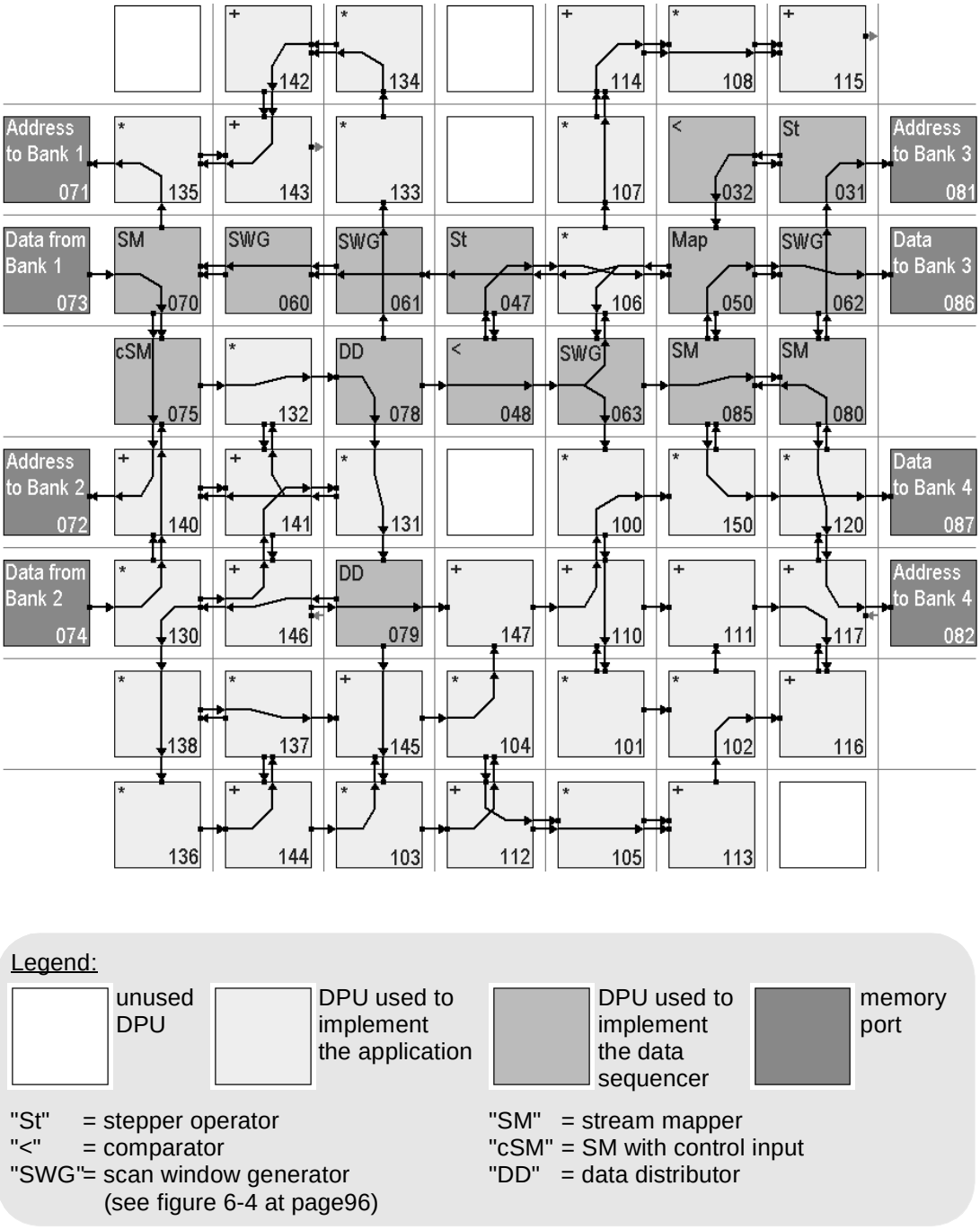
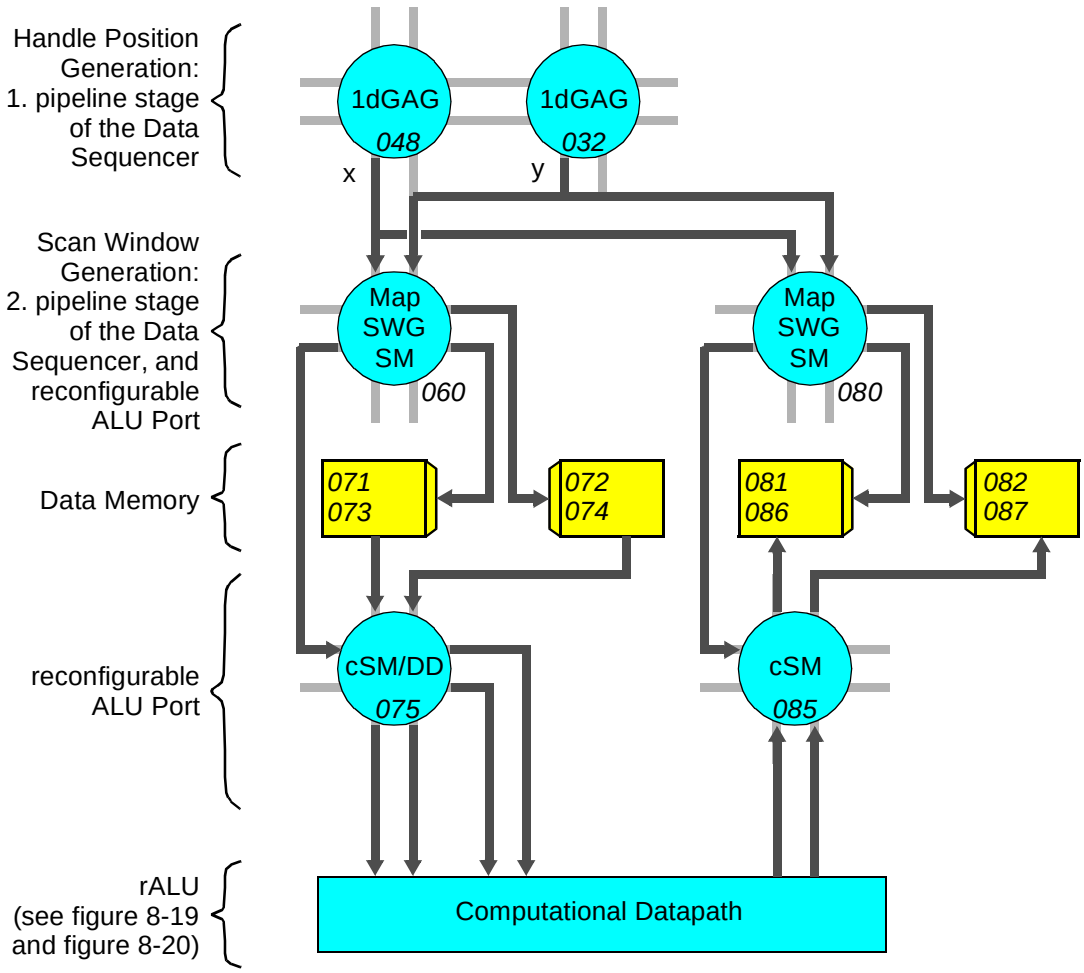


Figure 8-21: KressArray-3 mapping result for the linear filter implementation using DPUs of conventional functionality (see figure 8-18 at page 210).

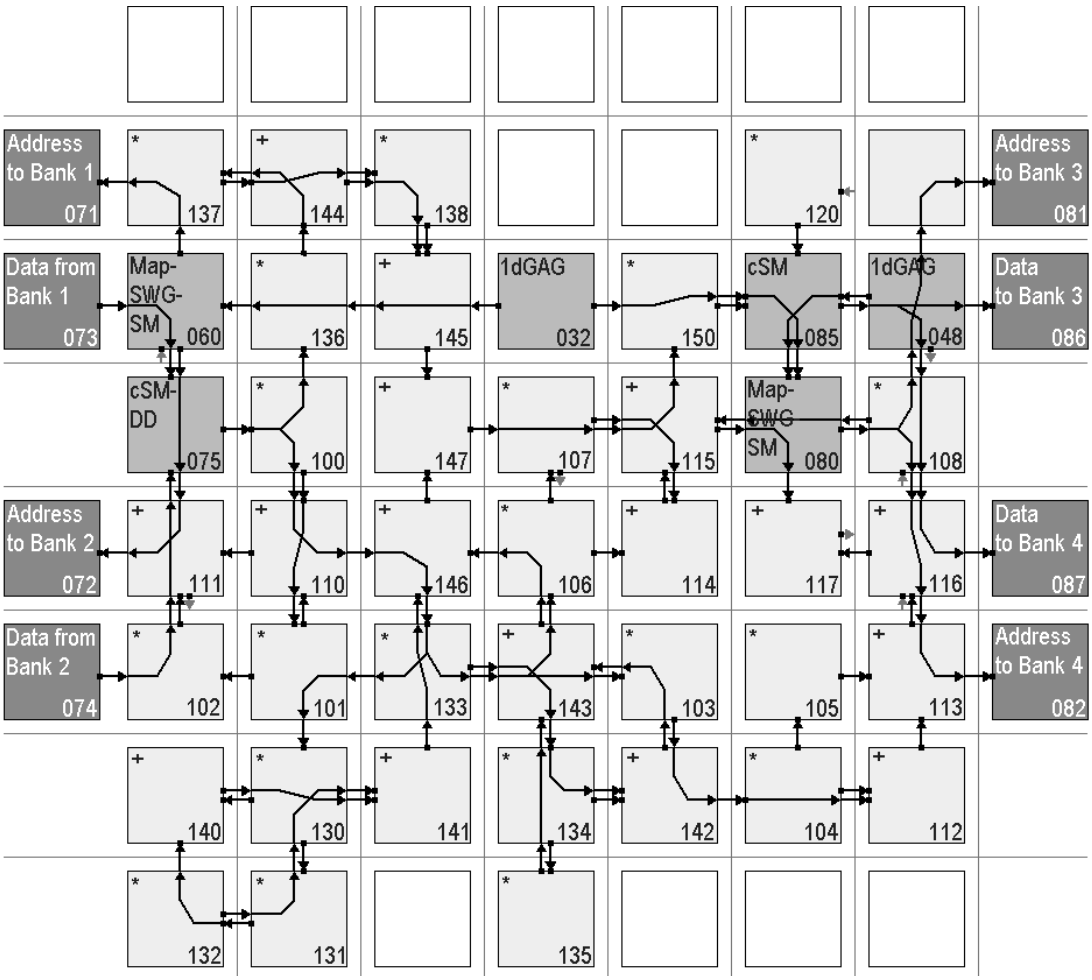
8.6.2 Example Implementation With Highly Integrated Multi-function Datapath Units

This example implementation utilizes the same KressArray architecture as the implementation above. But the granularity of the operators is chosen, that each operator may perform multiple arithmetic operations. This may be achieved by several ALUs inside the operator, or through sequential execution.



Legend:
"1dGAG" = 1-dimensional Generic Address Generator (see figure 6-11 at page112)
"Map/SWG/SM" = integrated memory mapper, scan window generator, and stream mapper
"cSM/DD" = integrated stream mapper with control input and data distributor
"cSM" = stream mapper with control input

Figure 8-22: Linear filter implementation for the KressArray-3 with multi-function DPUs. Cursive numbers indicate the operator and memory port numbers in the layout, pictured in figur e8-23 at page 216.



Legend:

unused DPU

DPU used to implement the application

DPU used to implement the data sequencer

memory port

"1d GAG" = 1-dimensional Generic Address Generator (see figure 6-11 at page 112)
"Map/SWG/SM"= integrated memory mapper, scan window generator, and stream mapper
"cSM/DD" = integrated stream mapper with control input and data distributor
"cSM" = stream mapper with control input

Figure 8-23: KressArray-3 mapping result for the linear filter implementation using multi-functional DPUs (see figure 8-22 at page 215).

The overall design of the linear filter implementation is shown in figure 8-22 at page 215. It consists of the following components:

- the handle position generator is implemented by two 1-dimensional Generic Address Generator DPUs ("1d GAG"). For internal implementation details see figure 6-11 at page 112,
- the handle positions are processed by a "Map/SWG/SM" operator, which merges the x- and y-parts of the handle position, performs the scan window accesses, and stream mapping to parallel memory banks,
- four memory banks, two for the source data, and two for result data,
- the reconfigurable ALU Port is build of a combination of a data distributor and a stream mapping operator with control input ("cSM/DD"), and a "cSM" operator, and
- the computational datapath. Its implementation is the same as in section 8.6.1 at page 209.

The mapping result generated by the MA-DPSS (see [HHH99a] or [HHH00]) is shown in figure 8-23 at page 216. 6 of 42 DPUs were used for the data sequencer implementation.

8.6.3 The Linear Filter Example Mapping Results

The mapping results for both DPUs types are shown in figure 8-24 at page 218. The implementation with multi-function DPUs requires less operators and also fewer routing resources. Even though the computational datapath has not been adopted to the higher functionality of multi-functional DPUs. Further the figures demonstrate, that the data sequencer requires fewer hardware resources than the rALU implementation and fits well into the same device.

8.7 Chapter Summary

This chapter has introduced an image processing example to demonstrate the benefits of the presented data sequencing concept and the optimization techniques based on it. Each software level optimization has been discussed individually and its effects have been described. For the application example the hardware level optimizations already provide a remarkable speed-up. In combination with the software level techniques the number of memory cycles has been reduced by the factor 11,3. Figure 8-13 at page 203 summarizes the speed-up figures at a glance.

A data sequencer performance analysis for the MoM-PDA data sequencer has shown that the accelerator performance can not always be improved in the same degree. Besides some overhead of the scan window generator during the generation of the

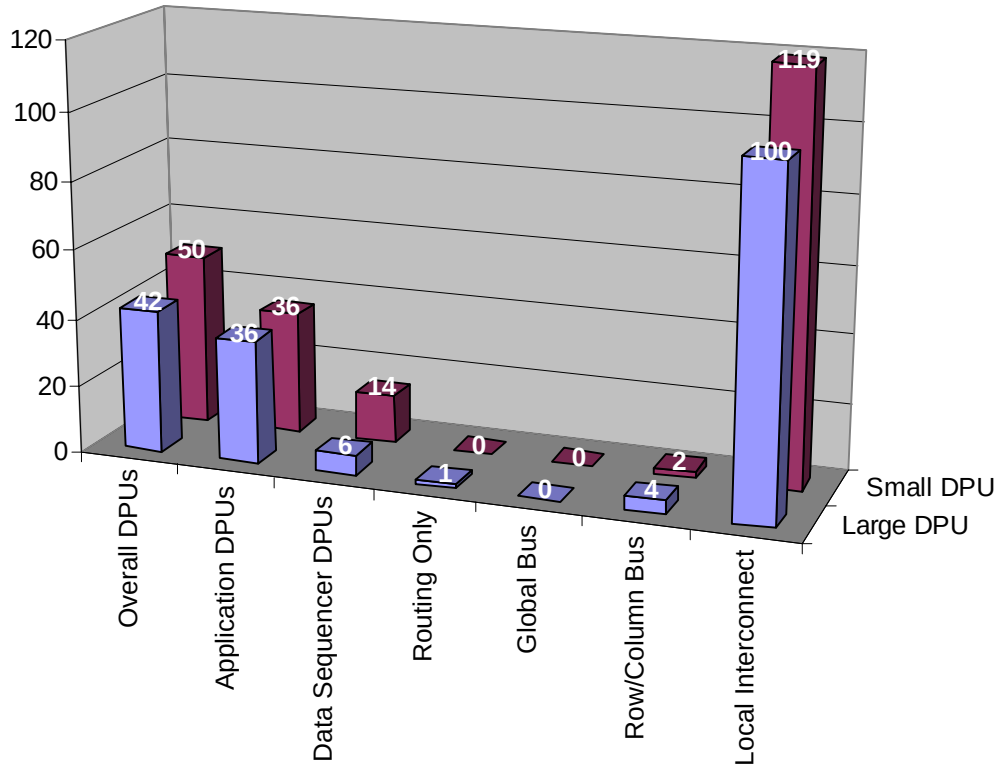


Figure 8-24: The linear filter mapping results.

memory cycles, the generation of the handle position requires a minimum number of memory cycles. An optimum handle position generator would provide a new handle position within one clock cycle, but the MoM-PDA data sequencer is a limited CPLD-based implementation. A remarkable fact is, that the optimization of the example application has also reduced the data sequencer configuration time by the factor 1,8. The reason for this improvement is the utilization of burst accesses.

The application example has also been implemented using the KressArray as target technology for the data sequencer as well as for the computational datapath. It has been shown, that the number of utilized DPUs can be efficiently decreased by using an application specific data sequencer implementation. Furthermore multi-function DPUs result in an additional reduction of DPUs needed for the data sequencer implementation.