7. Data Sequencer use for Higher Memory Bandwidth

A design study has been carried out to estimate the chip area for a KressArray cell [HHH00]. Assuming a 0.18 µm CMOS process, the routing resources of one cell in an 18 bits wide DPU architecture with 8 nearest neighbor ports (2 per side, see figure 2-9b at page 16), 2 row and 2 column buses need 4 metal layers of an area of approximately 0.06 mm^2. As the computational resources require less chip area and become faster and faster, the time data can be provided becomes increasingly important. Therefore the optimization of the memory communication presented in this chapter focuses on the reduction of memory cycles by a skillful application of memory access optimizations.

In order to improve the memory bandwidth in using the data sequencer concept presented in chapter 6, a dedicated memory architecture will be introduced in this chapter. On the basis of this memory architecture a methodology to optimize the memory throughput will be explained. If viewed individually, some techniques used here are already known or derived from known optimizations for conventional computers. But the optimization method presented here applies a combination of multiple techniques and makes them usable for reconfigurable computing. It exploits the flexibility of reconfigurable hardware to adapt and improve conventional techniques. In the following the coherence of the available methods will be elucidated.

The Memory Architecture

Already known from the MoM family and others like e.g. the VSP (see chapter 4) is the 2-dimensional organization of data memory (2d memory). It has the advantage of a good visualization of data structures and access sequences. But the 2-dimensional memory organization implies a data coherence problem when mapped into the linear address space.

The 2d memory will be accessed by the 2-level data sequencing method introduced in section 6.1.2 at page 92. A scan window is used to access the data needed in one computation step. Applications are computed by moving the scan window over the 2-dimensional data map. If successive scan window positions overlap during movement, the data is not read again from the memory, but will be stored in a cache-like register file called smart interface.

To provide a platform for memory bandwidth improvements, a multi bank memory system will be introduced, which enables concurrent memory accesses. Because of its 2-dimensional organization, this memory system will be called 2-dimensional multi bank (2d-MB) memory. A special row-major mapping scheme to map the 2d-MB memory into the linear address space of conventional memory devices has been developed.

A further speed-up will be achieved by the utilization of burst mode memories. The final memory architecture will be a 2-dimensional multi bank burst mode (2d-MBB) memory.

Problems of Throughput Optimization in Using 2d-MBB Memory

The throughput of 2d-MBB memory will be optimized in using three methods:

- modification of the application by a special loop unrolling transformation,
- modification of the storage scheme of the application data, and
- scheduling of the data accesses.

In particular the storage scheme of data and scheduling of data accesses holds unforeseen complexity:

For the MoM-3 in combination with the KressArray-1 an optimum datapath array I/O scheduling is performed [Kre96]. Since the MoM-3 uses a 2d memory with scan window access, no further throughput optimization is performed. The datapath array I/O scheduling exploits the degrees of freedom, given by the data dependencies of the application. This scheduling is predominantly a classical scheduling problem, wherefore the area of high level synthesis provides an abundant number of methods.

Interleaved memory systems access parallel memory banks in a fixed sequence. During optimum accesses to an interleaved memory system, a trigger event ripples through the banks in a fixed sequence. This method requires an alignment in time and space through all banks. To exploit the speed-up potential of interleaved memory the storage scheme must provide data locality through banks and the access scheduling must generate an access sequence through the memory banks in a predefined order. The potential parallelism of interleaved memory may only be exploited with a beneficial storage scheme. This is a well known problem, which is expatiated in the area of supercomputers.

Similar to interleaved memory, burst memory requires also a fixed storage scheme (locality of data) and access sequence, but in a single memory bank. This changes the boundary conditions for the storage scheme. Burst accesses reference a sequence of words within a row of memory cells. The optimization effect is the same as for interleaved memory. The difference is, that only one memory bank is needed.

Burst mode memory can further be used for an additional level of parallelism: several burst mode memory banks (2d-MBB memory). This kind of memory supports the application of two optimization methods simultaneously: burst mode and concurrent access to interleaved banks. While interleaved memory as well as burst mode may be applied individually to a 1-dimensional memory architecture, the simultaneous application of both methods requires an enhanced memory organization. This is also a challenge in order to find a good storage scheme and access sequence.

A further dimension in memory bandwidth optimization are scan window based or comparable memory architectures as known from the map-oriented machines (see chapter 4 and chapter 5). Here consecutive accesses to the same data are avoided by a hardware implementation of the scan window. This is in contrast to microprocessors, where a systematic prevention from multiple accesses to the same data is not possible. The spaghetti style access behavior of microprocessors may only be optimized by intensive manual code inspection, or an approximation is achieved by cache structures.

Chapter Overview

In the following this chapter will present a 2d-MBB memory architecture with window based access. For this architecture his chapter will explain a memory access optimization algorithm (section 7.5 at page 179), which includes the following methods:

- loop transformations (section 7.2 at page 162),
- modification of storage schemes (section7.3 at page165), and
- scheduling of memory accesses (section7.4 at page174).

7.1 The Memory Architecture

The memory architecture presented in this chapter is based on a dedicated 2-dimensional memory organization. Compared to 1-dimensional memory (see figure 7-1a), a 2-dimensionally organized memory (figure 7-1b) allows a good visualization of data structures, especially for applications operating on 2-dimensional data sets e.g. like in image processing. Also the address sequences and the storage schemes for interleaved memory access are easily depictable. Additionally the 2-dimensional organization will enable the synergistic combination of burst memories

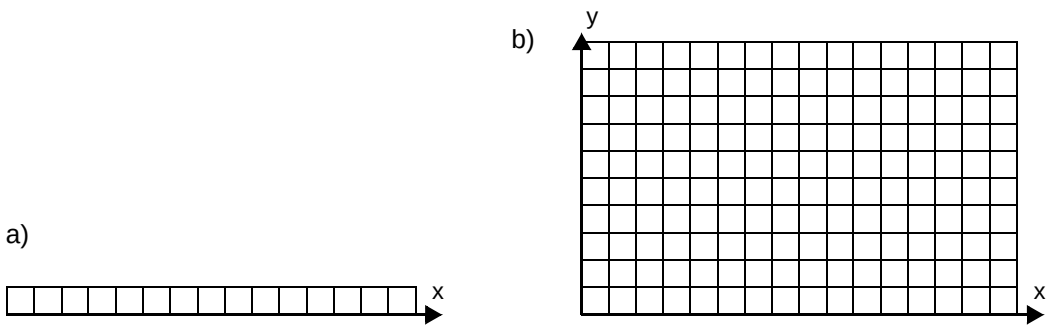

Figure 7-1: Memories of different dimension:
(a) 1-dimensional, and
(b) 2-dimensional.

and concurrent access to parallel memory banks. This thesis shows that 2-dimensionally organized memory is the basis for a novel memory access optimization method.

Two-dimensional data memory is defined as follows:

Definition 7-1: 2-dimensional Memory

Each memory location of a 2-dimensionally organized memory has two addresses: an x- and y-address. To access data, both addresses must be applied.

7.1.1 Row Major Mapping

In using a traditional row major mapping scheme the 2-dimensional memory is not implemented physically but is built up with conventional 1-dimensional memory devices. One reason for this may be, that there is no 2-dimensional memory available as defined in definition 7-1. But later in this section will be shown, that the mapping of 2-dimensional memory to 1-dimensional memory devices is also a source for optimization techniques.

In general, if a 2-dimensional memory is mapped to a single 1-dimensional memory device, row major mapping (see [HB85] or [PH90]) is applied as shown in figur e7-2. Row major means, that the memory is mapped row by row from the 2-dimensional address space into the linear address space.

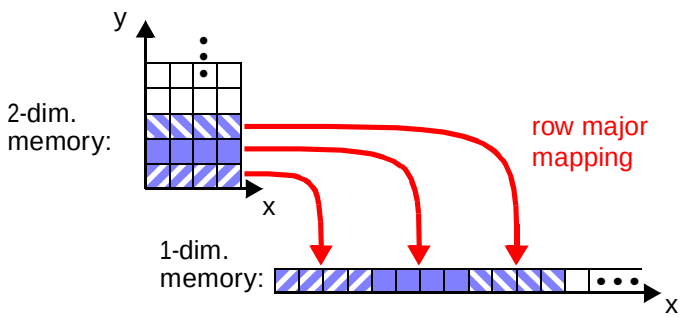

Figure 7-2: Row major mapping (see [HB85] or [PH90]) of 2-dimensional memory to 1-dimensional memory.

7.1.2 Parallel Memory Banks

By simply mapping a 2-dimensionally organized memory in row major style to a physically 1-dimensional memory, the memory communication performance will not be improved. In this case the utilization of 2-dimensional memory will only simplify

the implementation of applications operating on 2-dimensional data sets because of the better visualization. But 2-dimensional memory has the potential to simplify the implementation of a computing machine operating on parallel memory banks.

Usually parallel memory banks in computing systems are totally isolated. They use separate address generators, address lines and have their own address range. Such computing systems often struggle with data consistency, data routing and a bottleneck (see [AG94], [Kri89], or [Mor94]), which occurs, if multiple data sinks require data of one memory bank:

- Data consistency problems occur, if the same data is located in multiple memory banks, and manipulated by two different functional units concurrently.
- Data routing problems occur, if data stored in one memory bank is required by a functional unit, which has no direct data access to the memory bank. Expensive routing resources of the computing system are locked by such accesses.
- If multiple functional units operate on data stored in one of the parallel banks, a bottleneck appears, since the functional units lock out each other in accessing the same bank.

Having multiple parallel memory banks available, they can be used to form a 2-dimensional data memory by having the rows of the memory mapped to independent memory devices [HBH97b]. This mapping is illustrated in figure 7-3 at page 150. With this mapping data of the 2-dimensional memory, which is located in neighboring rows, can be accessed concurrently. Since there are usually more memory rows used by an application as parallel memory banks are available, multiple rows are mapped as described in section 7.1.2.1 to the same bank.

Each memory bank is a completely autonomous memory device. Each bank has its own address and data lines and is accessible in parallel to all other banks of the 2-dimensional memory. The memory banks are not interlaced by an interleaving scheme.

For applications, which are based on 2-dimensional data sets (e.g. image processing), data may be mapped to the 2-dimensional data memory without any reordering and data of neighboring rows is accessible in parallel. This memory organization scheme does not require complicated storage schemes to exploit concurrent data access.

7.1.2.1 Row Major Mapping With Parallel Memory Banks

Parallel memory banks efficiently exploit multiple access operations at the same time. But having one bank for each row of the 2-dimensional memory this method would require an enormous amount of hardware resources without any further optimization. For each row of the 2-dimensional memory datapaths between the memory banks and the functional devices would be required.

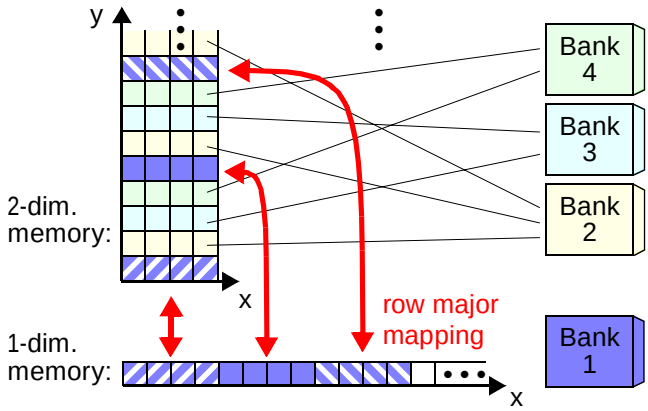

Figure 7-3: Row major mapping with parallel memory banks.

Row major mapping (see [HB85] or [PH90]) may also be applied, if parallel memory banks are used to implement 2-dimensional memory. As described above neighboring rows of the 2-dimensional memory are assigned to different memory banks. If there are n parallel memory banks available, the rows $y=1$ to $y=n$ are mapped to bank 1,2, ..., n. The row $y=n+1$ is again mapped to bank 1 and so on. In general, if there are n parallel memory banks, each row y is mapped to bank i if the following equation holds:

$$i = (y \bmod n)+1 \qquad \textit{(Eq. 7-1)}$$

To simplify the mapping of the rows of the 2-dimensional memory to the parallel memory banks, the number of the parallel banks n is restricted as follows:

$$n = 2^k, \forall k \in \{0, 1, 2, 3, \ldots\} \qquad \textit{(Eq. 7-2)}$$

Figure 7-3 illustrates row major mapping with $n=2^2=4$ parallel memory banks as described above.

7.1.2.2 The Dynamic Assignment of Relative Scan Window Positions to Parallel Memory Banks

A scan window, which accesses several rows of the 2-dimensional memory, is likely to access multiple memory banks (e.g. see figure 7-4 at page 151). Depending on the scan window movement, a scan window position may target different memory banks during movement. This fact demands a dynamic assignment of scan window rows to the parallel memory banks (see figure 7-4b at page 151). Since the number of parallel memory banks is $n=2^k$ ($k \geq 0$), the assignment of scan window rows depends on the least significant bits (LSBs) of the y-address.

Obviously scan window positions targeting different memory banks may be accessed concurrently. For low level sequencing a look-up table for each memory bank is needed. Depending on the LSBs of the y-part of the handle position the outputs of the look-up tables are dynamically assigned to the memory banks.

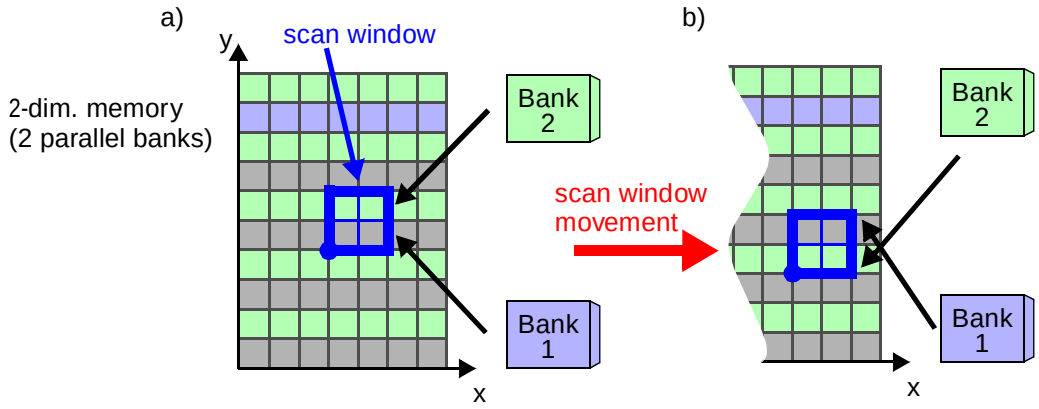

Figure 7-4: Dynamic assignment of scan window positions to memory banks during scan window movement:
(a) before scan window movement, and
(b) new assignment after scan window movement.

7.1.2.3 A Mapping Scheme to Solve Data Locality Coherence Problems

The above mapping scheme transforms the 2-dimensional memory into the linear address space. But by an awkward mapping from the 2-dimensional address space into the 1-dimensional address space, a data block will be torn into segments, which are interrupted by empty memory cells.

As explained above the 2-dimensional memory is mapped to one or multiple banks of linear memory. Two-dimensional memory addresses are composed of an x- and y-part with e.g. 16 bit for each (figure 7-5d at page 152). After generation of the x- and y-address parts, they are merged to form one physical memory address to access a linear memory. But simply merging two 16 bit x- and y-addresses to a 32 bit memory address would require an enormous memory size of 4 giga words with a fixed row length of 64k words for any application. But often some leading bits of the x and y addresses are unused. To reduce fragmentation caused by a fixed row length the output addresses are shifted according to the size of the actual data map, because not every application requires the complete address range of 16 bits for each dimension. This results in different data map sizes and shapes. The leading zeros of the x-address are the reason for this waste as can be seen in figure 7-5d at page 152. Therefore the higher bits of the address word are shifted to the lower positions until all unused lower zero bits are eliminated (figure 7-5e at page 152).

Figure 7-5a at page 152 pictures the data of an application example, which is mapped to 2-dimensional memory. If this data is mapped to a linear memory banks without elimination of the unused MSBs[1] of the x-address, the linear memory would be

1. MSB = most significant bit

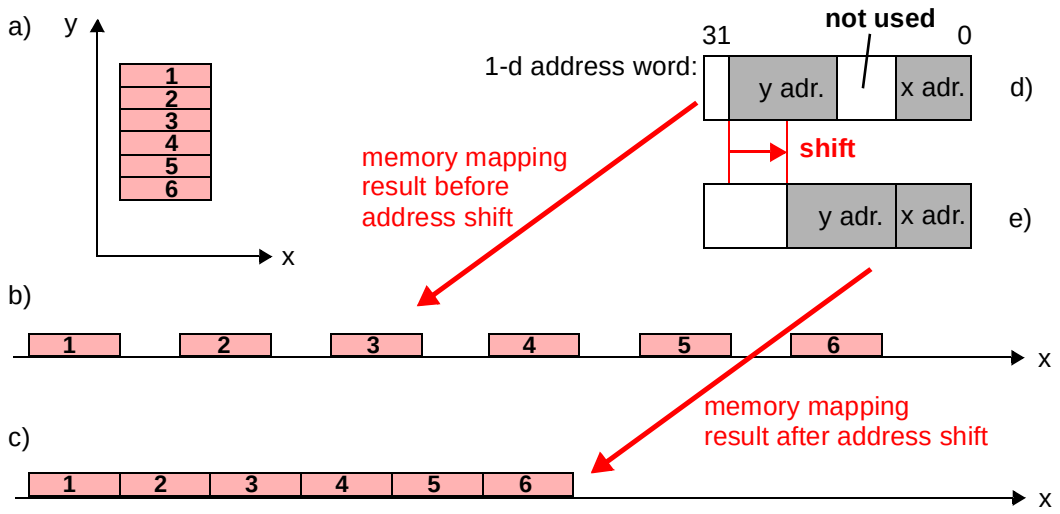

Figure 7-5: Illustration of row major address mapping for 2-dimensional array coherence:
(a) memory mapping of a 2-dimensional example
(b) to one linear memory
(d) without re-mapping by the memory mapper, and
(c) to one linear memory
(e) with re-mapping by the memory mapper.

fragmented as shown in figure 7-5b. This fragmentation can be avoided by an appropriate shift operation, which also results in a lower overall memory requirement (see figure 7-5c). Therefore a flexible mapping scheme is used, to map the used 2-dimensional address space of different x- and y-size onto the same physical memory system. The row major mapping method as pictured in figure 7-3 at page 150 is adjustable to the x-size of the data memory as required by a specific application. A hardware implementation of this mapping can be found in appendix D.2.3 at page 273.

7.1.3 Consequences of the Presented Memory Organization

The basis for the data sequencing method and the memory access optimization presented in this thesis is the 2-dimensional memory as described above. This 2-dimensional memory is mapped onto parallel 1-dimensionally organized memory devices using a flexible row major mapping scheme as explained in section 7.1.2 at page 148. The benefit of this method is that the data of an application can be mapped on parallel memory banks without a costly mapping scheme. Data of neighboring rows may be accessed concurrently while the problems of other approaches (see introduction of section7.1.2 at page148) are avoided.

Memory accesses to a single bank may be further optimized by other techniques like interleaved memory [Kog81] or burst accesses [Pri96]. Section 7.1.4.3 at page 158 focuses on this topic. Section 3.1 at page 25 presents several memory devices, which support burst access optimizations (e.g. SDRAM, DDR-SDRAM, or MDRAM).

7.1.4 The Hardware Level Support for Memory Access Optimization

In this subsection three hardware level optimization techniques are explained. The first access optimization is concurrent data access based on parallel memory banks (see also section 7.1.2 at page 148), second a cache-like access optimization is presented, and burst access modes of state of the art memories (see section 3.1 at page 25) improve the memory interface performance on hardware level. These techniques have also been briefly introduced in [HHH98c].

7.1.4.1 Concurrent Access to Parallel Memory Banks

The 2-dimensionally organized memory is mapped to commercial 1-dimensional memories. Therefore the 2-dimensional memory is cut in slices. Depending on the width of the data map linear memory segments are obtained, which can be appended and mapped to a linear data memory (row major mapping). Mappings to any number n of physical memories are possible, where $n=2^k$ and $k \geq 0$ (see equation 7-2 at page 150). Figure 7-3 at page 150 illustrates the mapping of a 2-dimensional data map onto $n=2^2=4$ parallel memory banks. For bank 1 row major mapping is shown. During scan window movement the positions of the scan window are dynamically assigned to the parallel memory banks (see also sect ion7.1.2.2 at page150).

This memory organization scheme causes that neighboring rows of a scan window are assigned to different memory banks, which can be accessed concurrently. In a coherent access space the number of access cycles $ac_{ConcurrentAccess}$ for m data accesses to n parallel memory banks is given in equation 7-3. It is assumed, that each memory access is performed in two steps (apply address and access data), whereas each step requires one access cycle. This models the timing of many synchronous DRAMs. For an example see the MDRAM in appendix C at page 253.

$$ac_{ConcurrentAccess} = 2 \cdot \left\lceil \frac{m}{n} \right\rceil \qquad \textit{(Eq. 7-3)}$$

The achieved speed-up depends on both, the number of parallel banks (n) and the number of accesses (m)

$$speedup_{par} = \frac{2 \cdot m}{ac_{ConcurrentAccess}} \approx n \text{, for a large } m. \qquad \textit{(Eq. 7-4)}$$

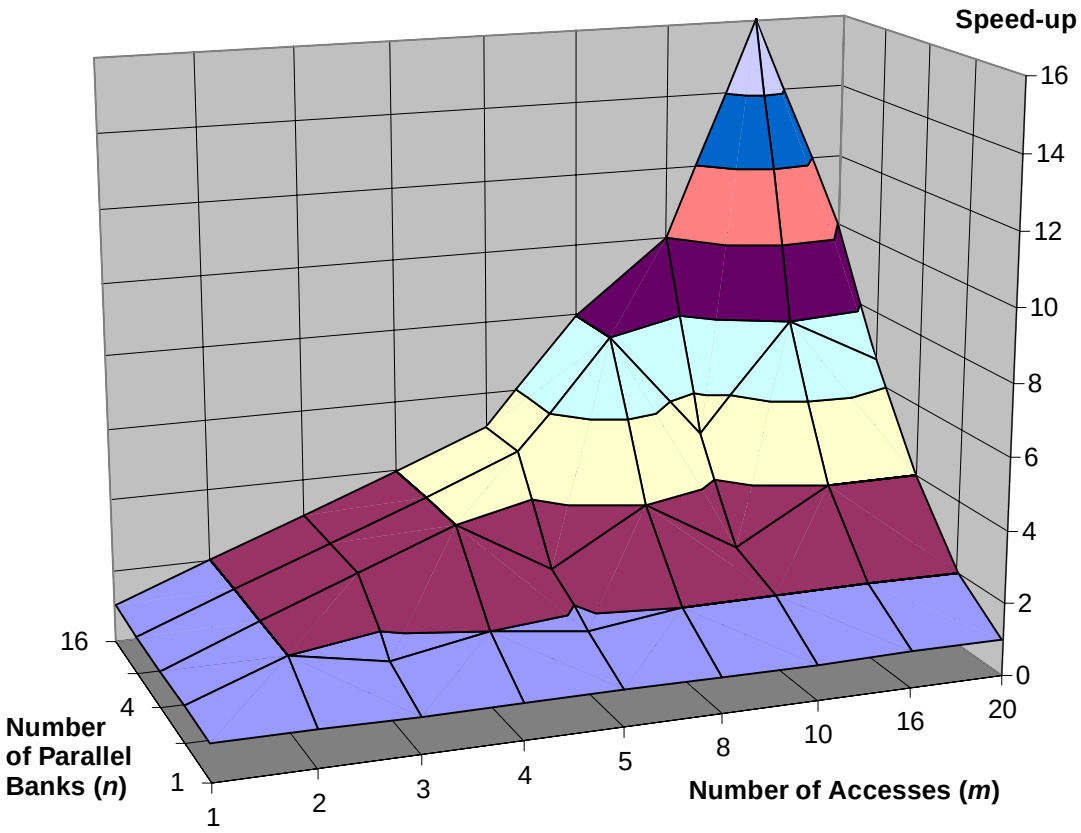

Figure 7-6: Illustration of the speed-up figures for the concurrent data access examples shown in table 7-1.

number of parallel banks (n) \ number of accesses (m)	1	2	4	8	16
1	1	1	1	1	1
2	1	2	2	2	2
3	1	1,5	3	3	3
4	1	2	4	4	4
5	1	1,67	2,5	5	5
8	1	2	4	8	8
10	1	2	2,5	5	10
16	1	2	4	8	16
20	1	2	4	6,67	10

Table 7-1: Speed-up figures ($speedup_{par}$) for concurrent data accesses.

If the accesses to the parallel banks are not coherent, i.e. one bank for example is not accessed, the speedup decreases. Also unbalanced accesses (e.g. 3 accesses to 2 parallel banks) lower the speed-up. Having more parallel memory banks as necessary, concurrent access results in no further speed-up and hardware resources are wasted. On the other side, having more memory accesses than parallel memory banks still results in good speed-up values. Table 7-1 at page 154 lists some speed-up figures for different combinations of *n* and *m*. These speed-up figures are also illustrated in figure 7-6 at page 154.

Because data needed in one iteration of a loop is mapped to parallel memory banks, parallel memory banks enable instruction level parallelism (ILP). Since multiple datapaths between memory banks and a parallel ALU are available, loop bodies can be computed concurrently on hardware level.

7.1.4.2 The Scan Window Overlap Optimization

An algorithm is executed by the movement of the scan window over the 2-dimensional memory, whereas the scan window holds the source and destination data. Depending on the order in which memory locations are accessed by the scan window a specific scan pattern results. Because of the regularity of memory accesses of loops regular scan patterns are achieved. Figur e7-7a shows a scan pattern called video scan, which often occurs e.g. in image processing.

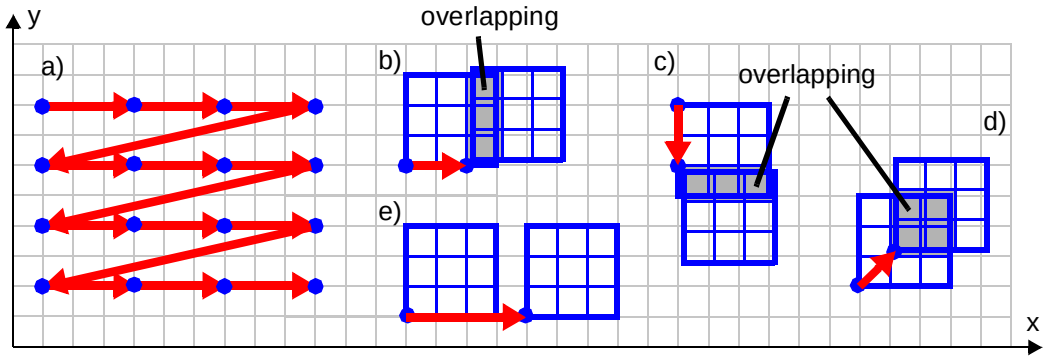

Figure 7-7: (a) scan pattern example,
(b) (c) (d) scan steps with scan window overlapping, and
(e) scan step without scan window overlapping.

Because of the scan window size may be larger than the scan pattern step width, scan window overlapping during computation occurs. Figure 7-7 also illustrates the different cases of scan window overlapping. While in figure 7-7e no overlapping occurs because of the scan step is longer than the scan window dimensions, the examples in figure 7-7b,c,d demonstrate overlappings in x-, y-, and x/y- directions. While non-overlapping scan window positions reference data, which was not used in the last computation steps, overlapping scan window positions access data used

directly before. This is the base of a deterministic *cache-like* memory access optimization. Because of the same scan window overlapping occurs at each scan window movement, the optimization can be implemented with little hardware effort. The overlapping positions can be figured out at compile time, when the scan window for the application is determined.

Because of traditional caches rely on the locality principle, cache hits can only be expected for an application [WH90], but not guaranteed. In contrast to this, the presented approach certainly provides the calculated reduction of memory accesses. This could be compared with a *hit-rate* of 100%. Memory accesses to scan window positions of the overlapping area can be saved completely.

Because of the number of overlapping positions depends on the application, a programmable *cache* is required. The the number of necessary registers varies. Therefore the flexibility of reconfigurable devices is exploited by implementing these registers within the reconfigurable datapath. The hardware implementation of this *cache* is called smart interface.

number of accesses inside a scan window (m_{SW}) \ number of overlapping scan window positions ($m_{overlap}$)	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	1	1	1	1	1	1	1
1		2	1,5	1,33	1,25	1,2	1,17	1,14	1,13	1,11	1,1
2			3	2	1,67	1,5	1,4	1,33	1,29	1,25	1,22
3				4	2,5	2	1,75	1,6	1,5	1,43	1,38
4					5	3	2,33	2	1,8	1,67	1,57
5						6	3,5	2,67	2,25	2	1,83
6							7	4	3	2,5	2,2
7								8	4,5	3,33	2,75
8									9	5	3,67
9										10	5,5
10											11

Table 7-2: Speed-up figures ($speedup_{overlap}$) of the scan window overlap optimization for some scan window examples.

The speed-up achieved by the smart interface optimization is the quotient of the number of accesses of the unoptimized scan window (m_{sw}) and the number of accesses without the accesses omitted by the scan window overlap optimization (m_{sw}-$m_{overlap}$):

$$speedup_{overlap} = \frac{m_{sw}}{m_{sw} - m_{overlap}} \qquad \textit{(Eq. 7-5)}$$

Table 7-2 at page 156 lists the speed-up for scan windows with a number of accesses between one and eleven. For this scan windows all possible numbers of overlapping positions are considered. The speed-up figures are also illustrated in figure7-8.

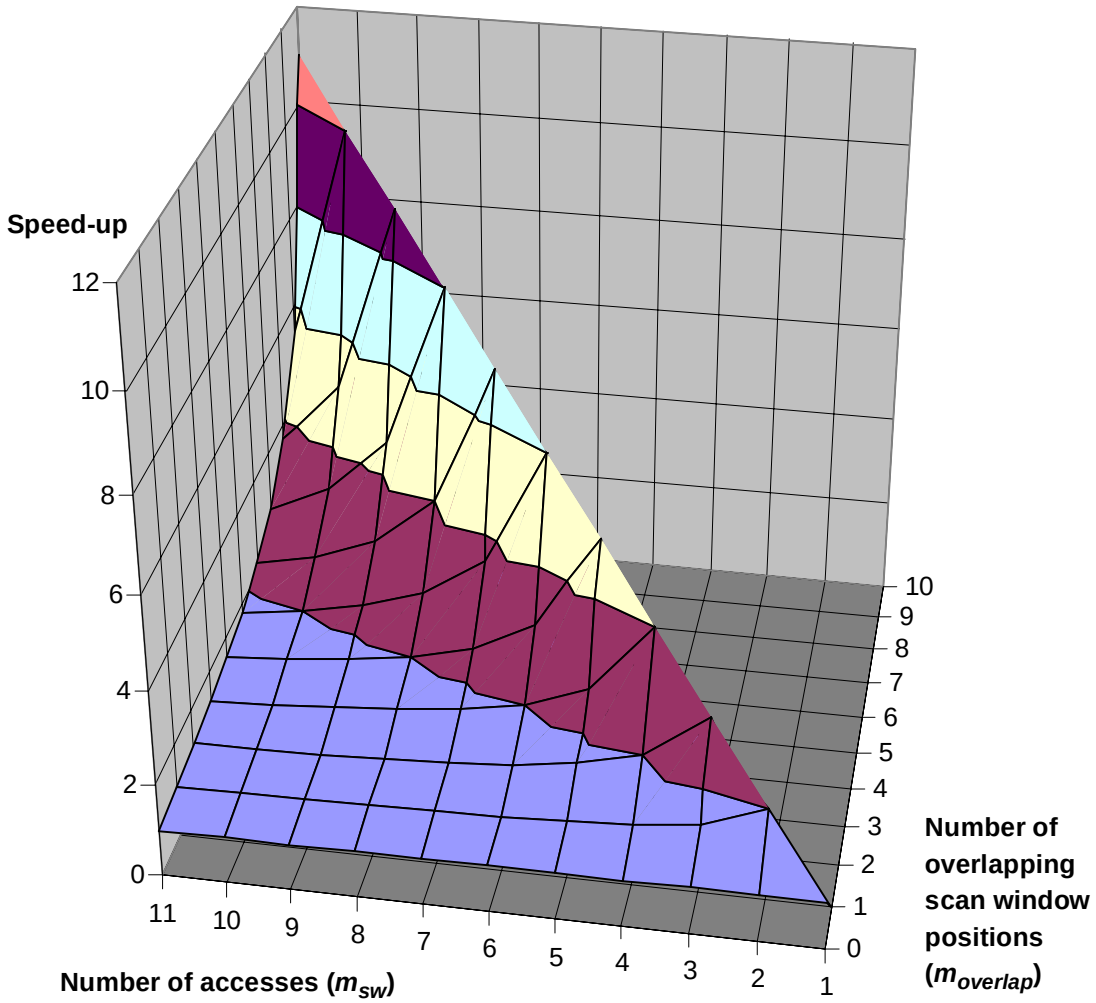

Figure 7-8: Illustration of the speed-up figures for the scan window overlap optimization shown in table 7-2 at page 156.

7.1.4.3 The Burst Access Optimization

Since modern memory devices a provide burst access mode, this optimization may also be adopted to 2-dimensional memory. Originally burst read operations where introduced for conventional computers to fill complete, fixed-width cache lines within only one memory access. Multibank DRAMs (MDRAM, see section 3.1 at page 25, or appendix C) allow variable length burst read or write operations.

Burst accesses are performed by setting a start address and the memory generates automatically accesses to the following memory locations until a stop-command is sent. The number of performed accesses is called the burst length (see figure 7-9). The burst is performed in a row of the memory matrix. Therefore the burst length is limited by the row length. For MDRAMs the maximum burst length is 32 words each of 32 bits (see section C.1 at page 253).

The technique of burst memory access is used inside a scan window, if accesses of the same kind (read or write) are located horizontally side by side. Figure 7-9 shows an example scan window with several burst accesses inside. Because of only an initial address has to be set, the memory accesses are performed faster, i.e. for the subsequent accesses the apply address step can be omitted.

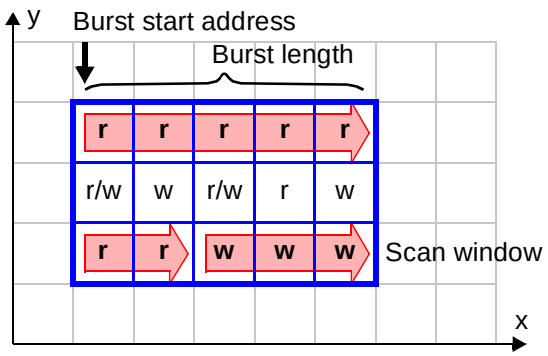

Figure 7-9: Burst accesses inside a scan window.

Usually for each memory access inside a scan window, an entry in the look-up table of the scan window generator (see section 6.1.3 at page 94) is required. While burst accesses reference multiple memory locations, they require also only one entry in the scan window generators look-up table. As a result burst accesses save configuration cycles.

For synchronous DRAMs like MDRAMs it takes the same time to apply an address as to perform one data access. The following formula calculates the number of access cycles for *m* memory accesses without burst:

$$ac_{NormalAccess} = 2 \cdot m \qquad \textit{(Eq. 7-6)}$$

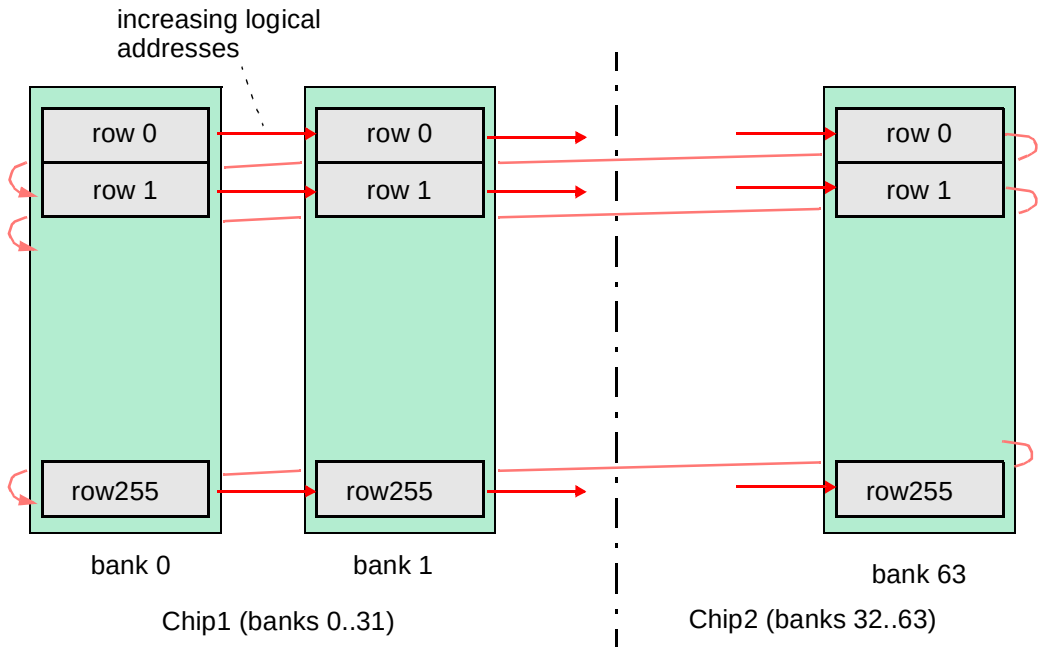

Figure 7-10: Bank interleave addressing scheme of the Map-oriented Machine with Parallel Data Access (MoM-PDA, see appendi xD.1 at page 262).

Equation 7-7 gives the number of clock cycles for the same accesses in one burst.

$$ac_{BurstAccess} = 1 + m \approx m, \text{ for large } m \qquad \textit{(Eq. 7-7)}$$

The speed-up for a given number of accesses m can be calculated as follows:

$$speedup_{burst} = \frac{ac_{NormalAccess}}{ac_{BurstAccess}} = \frac{2 \cdot m}{1 + m} \approx 2, \text{ for large } m. \qquad \textit{(Eq. 7-8)}$$

The speed-up for long burst accesses (large m) can be approximated as 2. This is can also be seen in table 7-3 and figur e7-11 at page160.

number of accesses (m)	1	2	3	4	5	10	20	$\rightarrow\infty$
speed-up	1	1,3	1,5	1,6	1,7	1,8	1,9	$\rightarrow$2

Table 7-3: Speed-up figures for memory accesses in burst mode, calculated under application of equation 7-8.

The equation7-7 and equation7-8 do not consider typical refresh cycles of DRAM or if a burst exceeds the row of a memory bank. In that case the burst has to be interrupted and continued in a new row. Figure 7-10 shows the bank interleave addressing scheme of the Map-oriented Machine with Parallel Data Access (MoM-PDA, see

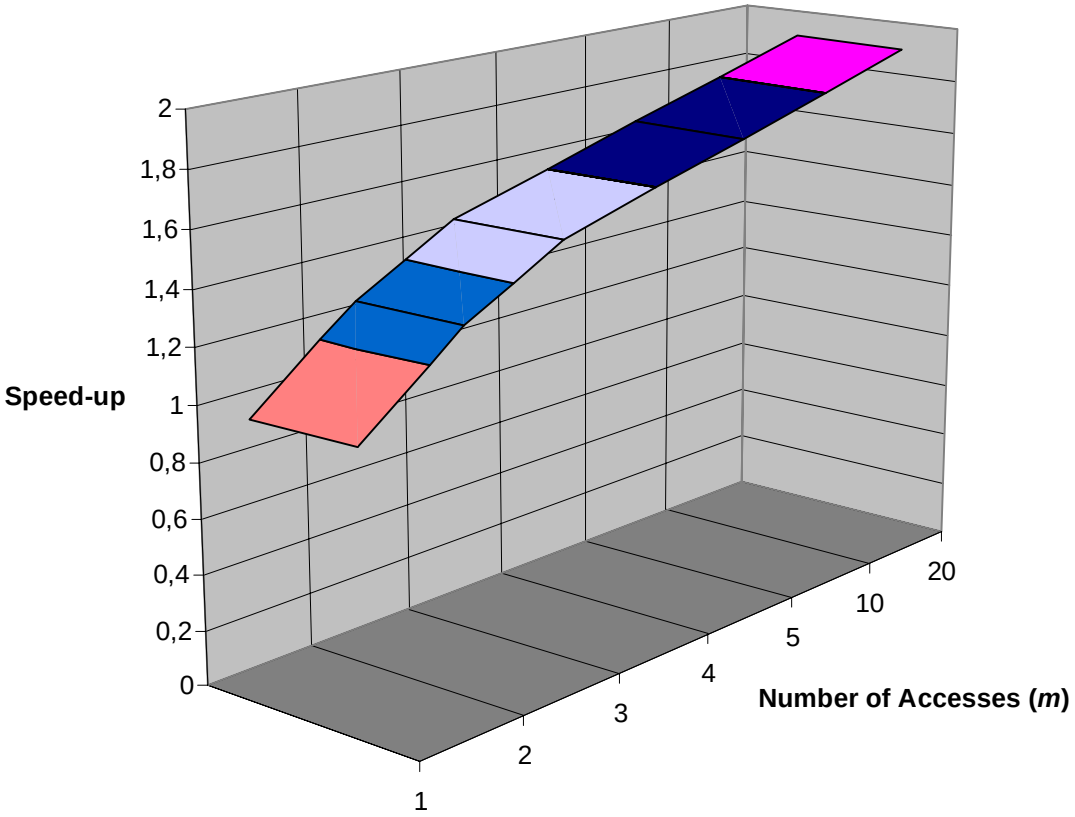

Figure 7-11: Illustration of the speed-up figures for the burst access optimization shown in table 7-3 at page 159.

appendix D.1 at page 262). Increasing bank address has higher priority than increasing row address. If a burst has reached the end of a row, it will be continued in the *same row of the next bank* and not in the *next row of the same bank*. This accelerates memory accesses, because activating of a row in a new bank can be done in parallel to accesses in the actual bank (for more details see [Bed98]). This is one of the most important benefits of MDRAM.

7.1.4.4 Summary of Hardware Level Optimizations

Three different techniques to accelerate memory accesses have been introduced:

- scan window overlap,
- concurrent access to parallel memories, and
- burst access mode.

These methods complement one another as they are effective in different directions. As defined in section 7.1.2 at page 148, parallel memory accelerates memory accesses in columns. Since burst accesses and concurrent accesses to parallel memory are orthogonal, burst accesses speed-up accesses in a row of the 2-dimensional memory. Scan window overlappings act in any direction of the 2-dimensional memory.

The three methods have also different degrees of optimization. This results in an order of priority based on this degree:

- Scan window overlapping (see section 7.1.4.2 at page 155) has the highest speed-up, because memory locations, needed several times during successive computation steps, are accessed only once. The degree of optimization increases with the overlapping area.
- Memory accesses to parallel banks (see section7.1.4.1 at page153) are performed concurrently. The degree of optimization increases with the number of parallel memory banks.
- Burst accesses (see section 7.1.4.3 at page 158) save the time to set the memory addresses but the accesses are performed sequentially. The speed-up increases with the burst length.

Figure 7-12 summarizes the affected direction and degree of speed-up of the hardware level optimizations. All these techniques may be applied simultaneously, because of the 2-dimensional data memory.

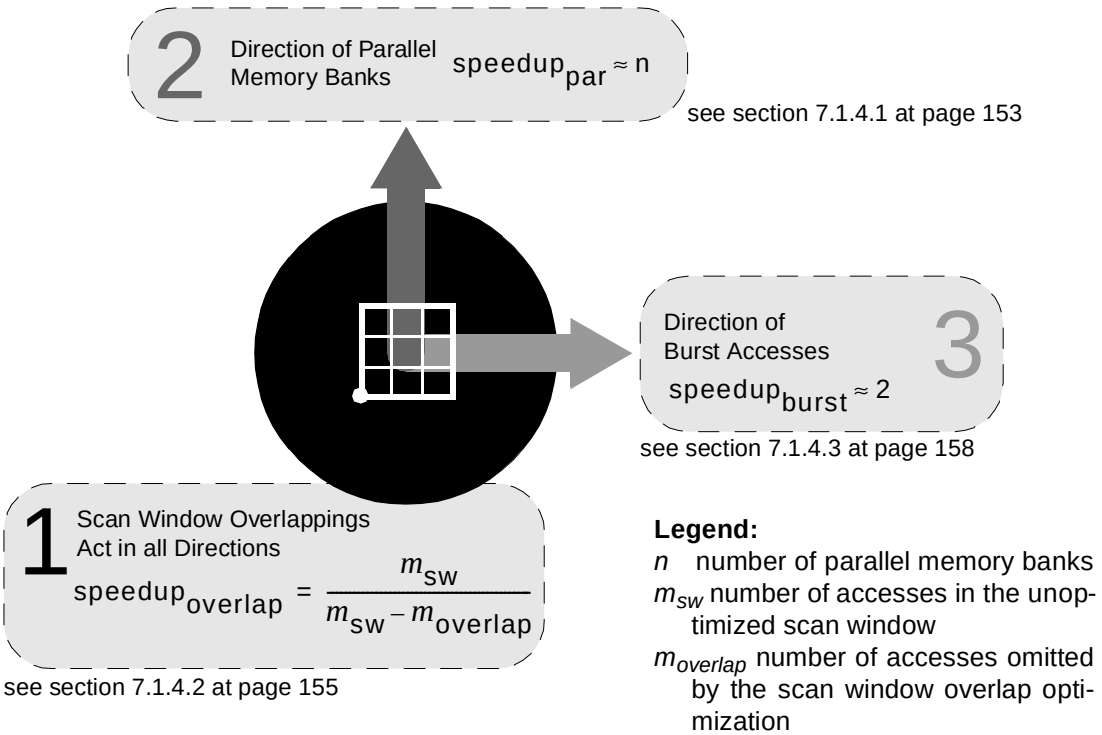

Figure 7-12: Summary of hardware level access optimizations.

7.2 Loop Transformations

Loop transformations are a well know technique to improve the utilization of hardware (see [Bec97] for reconfigurable accelerators or [Hwa93], [Lov77], [Mol93] for microprocessors). But one of these techniques can also be applied to improve the memory interface performance. Since loop unrolling improves the exploitation and the efficiency of the hardware level optimizations, it influences directly the memory interface speed.

Basically loop unrolling can only be applied up to a certain degree. Loop unrolling can only applied as long as the utilization of the reconfigurable datapath hardware resources is below 100%. A further limit is the capacity of the memory interface. But even if the memory interface is fully utilized, loop unrolling can increase its performance. Further loop unrolling can also decrease the utilization of the memory interface.

Loop Unrolling in General

The code transformation technique of loop unrolling increases the step width of single loop iterations and decreases therefore the total number of iterations by unrolling the loop body up to a certain factor. This factor is called unrolling factor. Thus multiple iterations of the original loop are executed in one iteration of the unrolled loop. An important condition for loop unrolling is, that the number of loop iterations is divisible by the unrolling factor. For more details see [Hwa93], [Lov77], or [Mol93].

In the context of scan pattern, loop unrolling causes modification of the scan pattern and the scan window. Then number of generated handle positions is decreased. Therefore the step width of the scan pattern and also the size of the scan window is increased. It can be distinguished between two types of loop unrolling; Inner Scan Line Loop Unrolling and Scan Line Unrolling. Mostly loop unrolling is the basis for further improvements by modification of the storage scheme.

7.2.1 Inner Scan Line Loop Unrolling

For Inner Scan Line Loop Unrolling the unrolling factor must be chosen, that he divides the number of scan steps of all scan lines of a video scan. Figure 7-13 at page 163 shows an example with an unrolling factor of 3. The size and shape of the unrolled scan window is modified that it covers the memory locations of as many consecutive scan windows as the value of the unrolling factor.

Depending on the direction the scan window increases, loop unrolling influences burst operations (in x-direction) and concurrent memory accesses (y-direction). The scan window overlap optimization is not touched. It is neither improved nor made worse.

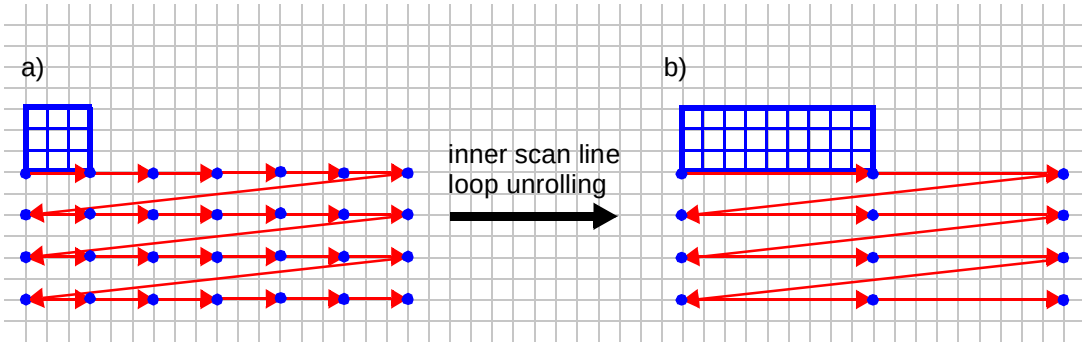

Figure 7-13: Scan pattern example:
(a) before inner scan line loop unrolling, and
(b) after inner scan line loop unrolling with unrolling factor 3.

7.2.2 Scan Line Unrolling

Scan Line Unrolling does not modify the number of scan steps in a scan line, but decreases the number of scan lines. Therefore the unrolling factor must divide the number of scan lines. Further Scan Line Unrolling can only be applied, if the number of scan steps is equal for all scan lines. Figure 7-14 shows an example with an unrolling factor of 2. The size and shape of the unrolled scan window is modified that it covers the memory locations of the scan windows of the same position of as many consecutive scan lines as the value of the unrolling factor.

Depending on the direction the scan window increases, scan line unrolling influences burst operations (in x-direction) and concurrent memory accesses (y-direction). Additionally Scan Line Unrolling can decrease the overall number of memory accesses. Since scan window overlapping can only be exploited between consecutive scan steps, scan window overlapping between scan lines can not be exploited. Scan

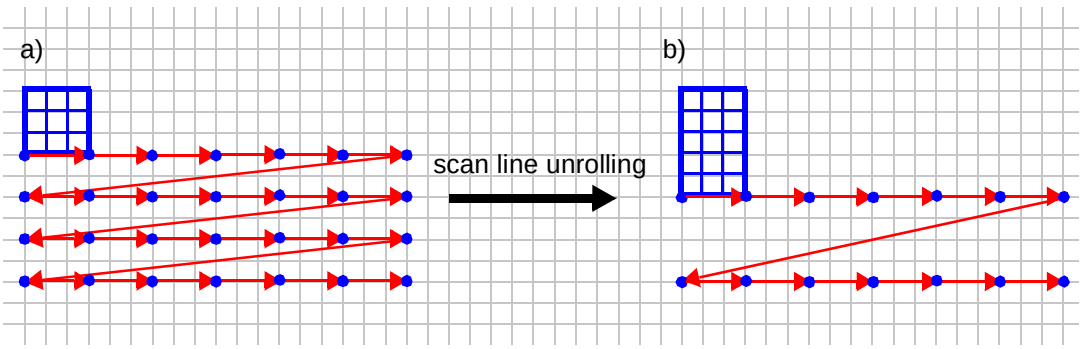

Figure 7-14: Scan pattern example:
(a) before scan line unrolling, and
(b) after scan line unrolling with unrolling factor 2.

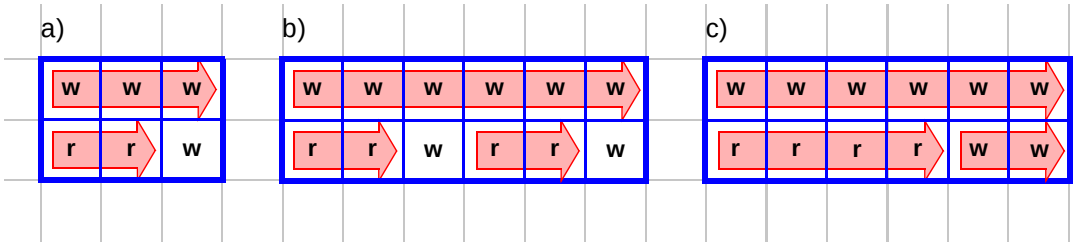

Figure 7-15: Scan window example with
(a) burst accesses inside,
(b) burst accesses after loop unrolling, and
(c) burst accesses after loop unrolling and modification of the storage scheme.

Line Unrolling joins neighboring scan lines. Memory accesses, which are repeatedly performed over several scan lines, can be saved. But this technique can not always be applied. Caused by data dependencies scan line unrolling is not always allowed.

Effects of Loop Unrolling on Burst Accesses

The effect of Inner Scan Line Unrolling and Scan Line Unrolling on burst accesses is the same. If loop unrolling acts in x-direction the length of burst accesses may be increased. Sometimes a good result demands a modification of the storage scheme.

As a result of loop unrolling the speed-up of a burst access is improved (see equation 7-8 at page 159). The overall load of the memory interface decreases because the number of address cycles decreases. That means, the overall number of memory cycles for this application is reduced.

Figure 7-15 illustrates the effects of loop unrolling on burst accesses with an example. The initial scan window (figure 7-15a) performs two burst accesses and one single access with a total of 9 memory cycles (3 times address, 6 times data). After unrolling with the unrolling factor 2 (figure 7-15b), the same data is sequenced with 8.5 memory cycles (half of 5 times address, 12 times data). Since there is no scan window overlap, the storage scheme can be modified as pictured in figure 7-15c. As a result the same application requires 7.5 memory cycles (half of 3 times address, 12 times data).

Effects of Loop Unrolling on Concurrent Data Accesses

The effect of Inner Scan Line Unrolling and Scan Line Unrolling on concurrent data accesses is the same. Loop unrolling increases the exploitation of parallel memory banks significantly, if the number of parallel banks is higher than the initial scan window height (see section 7.1.4.1 at page 153). Loop unrolling may even increase the performance of concurrent accesses, if all parallel banks are already utilized. This is the case when concurrent accesses to parallel banks are not balanced (see table 7-1 at page 154: e.g. if an application with 5 accesses to 2 parallel banks is processed with an unrolling factor 2, the speed-up is raised from 1,66 to 2).

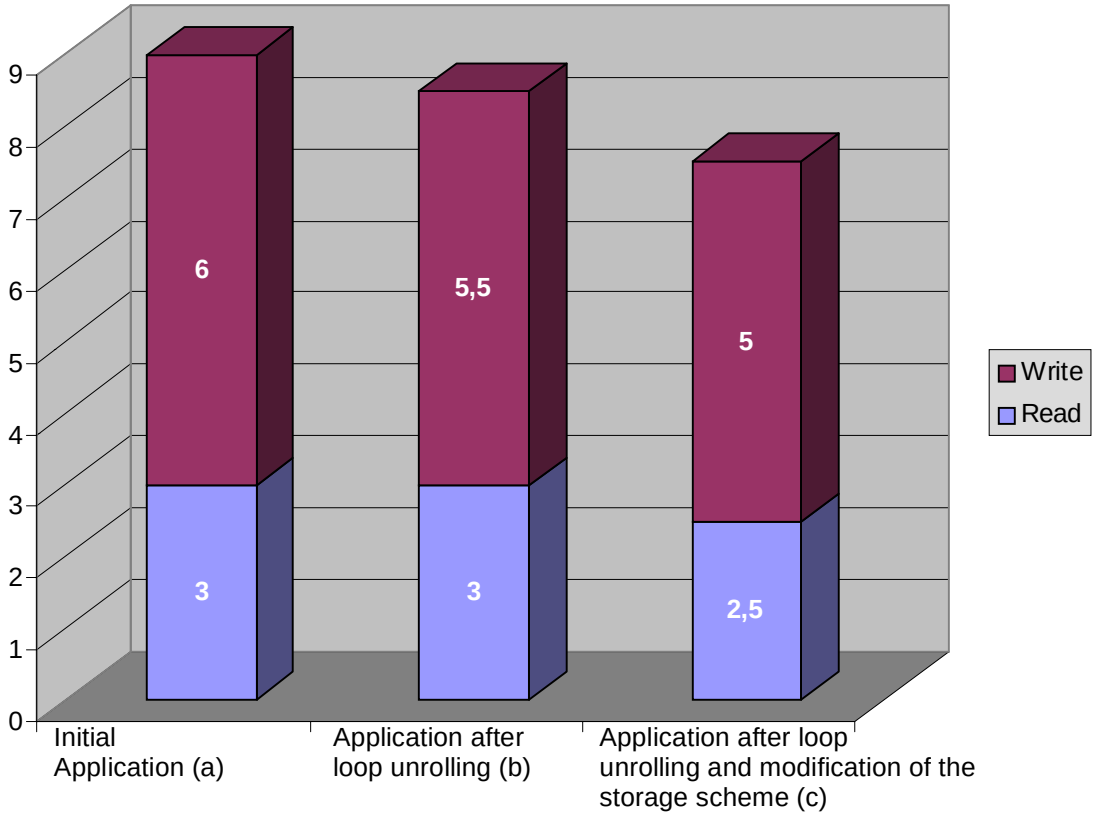

Figure 7-16: Number of memory cycles of the loop unrolling application example in figure 7-15 at page 164.

7.3 The Modification of Storage Schemes

The storage scheme in which application data is mapped to the memory holds the potential to improve the memory interface performance (other information on re-ordering of data can be found in [Law75]). Therefore in this thesis two different types of storage scheme modification will be presented:

- Low level storage scheme modification targets the low level data sequencing. The accesses inside a scan window are modified by a fine grain re-ordering of the data. Obviously this modification affects the complete data map.
- High level storage scheme modification affects the high level data sequencing by a coarse grain re-arrangement of the complete data set. Here operations like rotation, mirroring, and compressing are performed on the data map.

The two types of storage scheme modification will be explained in the following subsections. Generally the main goal of the storage scheme modification is to improve the memory communication performance on the basis of the 2d-MBB memory.

Additionally high level storage scheme modification has the potential to save memory space and data sequencer resources. But in general it depends on the application if storage scheme modification is applied and what type of modification is performed. Also the achieved degree of optimization depends heavily on the application.

7.3.1 The Low Level Storage Scheme Modification

The low level storage scheme modification re-orders the data inside a scan window. Since all computation data is accessed by a scan window, this modification affects the complete data map. But data is only re-ordered locally, neither the scan pattern, nor the size and shape of the scan window are modified. The goal of low level storage scheme modification is the reduction of memory cycles. This is achieved by an equable distribution of memory accesses of the same type (read or write) to the parallel memory banks to enhance the speed-up achieved by concurrent accesses, and by the elongation of burst accesses.

If there is no scan window overlapping between successive scan window positions or between scan lines, all data inside a scan window may be rearranged in any way. If scan window overlappings occur, arbitrary rearrangement is only allowed in the non-overlapping area of a scan window. If a rearrangement to an overlapping area is performed, the same re-arrangement must be performed to all areas of the scan window, which overlap with this area.

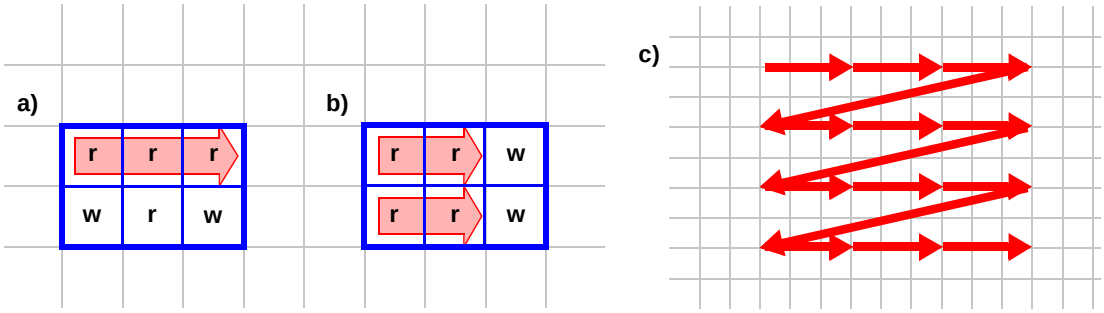

Figure 7-17: Low level storage map modification example:
(a) scan window before modification,
(b) scan window after low level storage map modification, and
(c) example scan pattern.

Figure 7-17a shows a scan window example and figure 7-17c the according video scan. No scan window overlapping occurs during scan window movement. Therefore all accesses inside the scan window may be rearranged in order to improve the access time. Since the speed-up of the parallel memory bank optimization is higher than the speed-up of burst accesses (see figure 7-12 at page 161), first the scan window content is distributed equally to the parallel memory banks. Therefore the accesses inside the

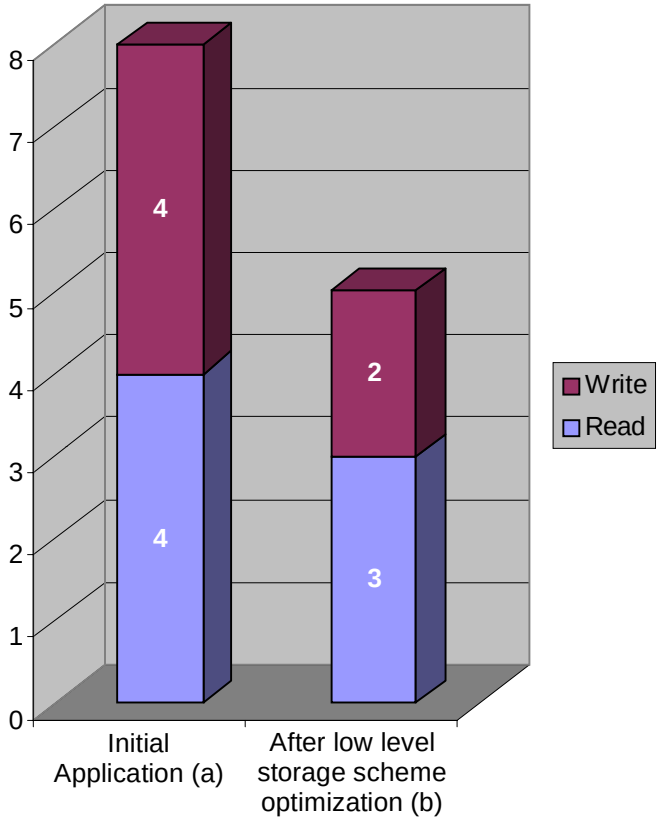

Figure 7-18: Number of memory cycles of the storage scheme optimization example in figure 7-17 at page 166.

scan window are grouped in read and write accesses. After that scan window positions are exchanged in order to have approximately the same number of read accesses and the same number of write accesses for each of the parallel memory banks. Sometimes the accesses can not be totally uniformly distributed because the number of parallel memory banks is not a divisor of the number of memory accesses. For example this is the case, if there are three accesses of the same type and two parallel memory banks.

After the assignment of read and write accesses to the parallel memory banks, i.e. to the scan window rows. The accesses inside the scan window rows are sorted. First all read accesses are located inside a scan window row. After that all compound read and write accesses are placed, followed by the remaining write accesses. Doing so the complete scan window row may be read by only two burst accesses.

Figure 7-17a at page 166 shows a scan window example and figur e7-17c at pa ge166 the according scan pattern. During scan window movement no scan window overlapping occurs. The upper scan window row contains three read accesses and to lower scan window row 1 read access. An uniform distribution of the read accesses is achieved by an exchange of one of the upper read accesses with one of the lower write accesses. This also equably distributes the write accesses.

To achieve long burst accesses the read accesses are moved to the beginning of the scan window rows. Figure 7-17b at page 166 pictures the optimized scan window. Figure 7-18 at page 167 compares the memory cycles of the scan windows under consideration of all hardware level optimizations. The speed-up of this example modification is obtained by the division of the initial number of memory cycles by the final number of memory cycles:

$$speedup_{LowLevelExample} = \frac{8}{5} = 1,6 \qquad \textit{(Eq. 7-9)}$$

7.3.2 The High Level Storage Scheme Modification

The high level storage scheme modification manipulates the complete data map and the scan pattern of an application. On this level three different operations may be performed to the storage scheme:

- transformation of the data map,
- exchange of x and y, and
- compression of the data map.

It depends on the application, if one or multiple of this operations can be applied. The high level storage scheme modification cause one or several of the following improvements:

- Save data memory.
- Save hardware resources for the data sequencer design (see also section 6.5.3 at page 133).
- Save memory cycles through better exploitation of hardware level optimizations.
- Save memory cycles through better exploitation of software level optimizations.

The three operations for high level storage scheme modification and their effects will be explained in the following.

The Exchange of x And y

The exchange of x and y modifies a complete application by the exchange of all x and y parameters. This operation is equal to mirror the complete application at the straight line, which bisects the angle between the x- and y-axis (see the straight line *g* in figure 7-19a at page 169 or figure 7-19b at page 169). The transformation matrix for this operation is given in equation 7-10 at page 169. The result of this operation is, that

burst- and parallel memory accesses are switched. The exchange of x and y is performed when accesses to the parallel memory banks are unbalanced, and/or to lengthen burst operations (see section7.1.4 at page153).

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad \textit{(Eq. 7-10)}$$

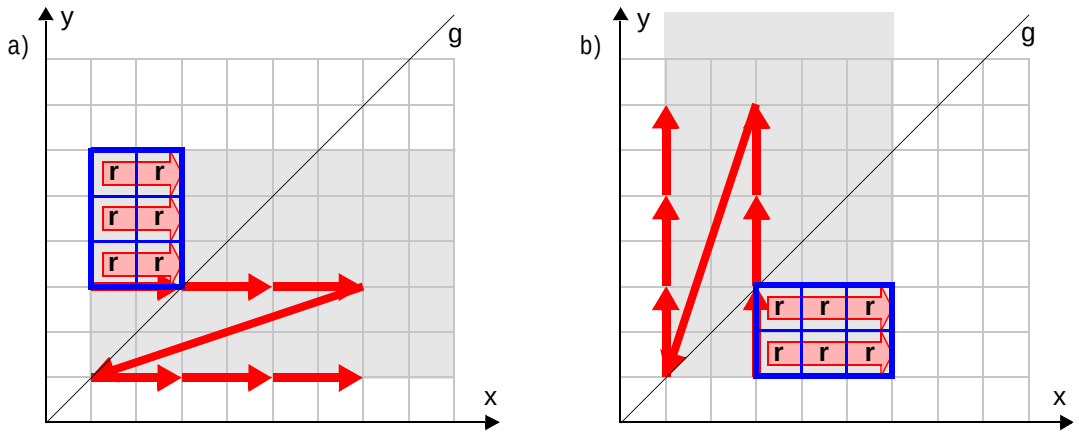

Figure 7-19: High level storage scheme modification: exchange of x and y:
(a) before modification, and
(b) after modification.

The basis for the decision, if the exchange of x and y should be performed, is the execution time of the original and the modified scan window. Therefore for both scan windows low level storage scheme modification and scheduling is performed. If the original scan window required less memory cycles the exchange of x and y is not performed. Otherwise the complete application is processed by the transformation matrix given in equation7-10.

Figure 7-19a pictures an application example, where accesses to two parallel banks are unbalanced. The modified application (see figure 7-19b) has longer bursts and the accesses to the parallel banks are balanced.

The memory cycles of the original scan window (see figure 7-20 at page 170) have been improved by the following speed-up value:

$$speedup_{HighLevelExample1} = \frac{6}{4} = 1,5 \qquad \textit{(Eq. 7-11)}$$

The exchange of x and y only reduces the number of memory cycles through better exploitation of hardware and other software level optimizations. It does not decrease the amount of used data memory and hardware resources for the data sequencer design.

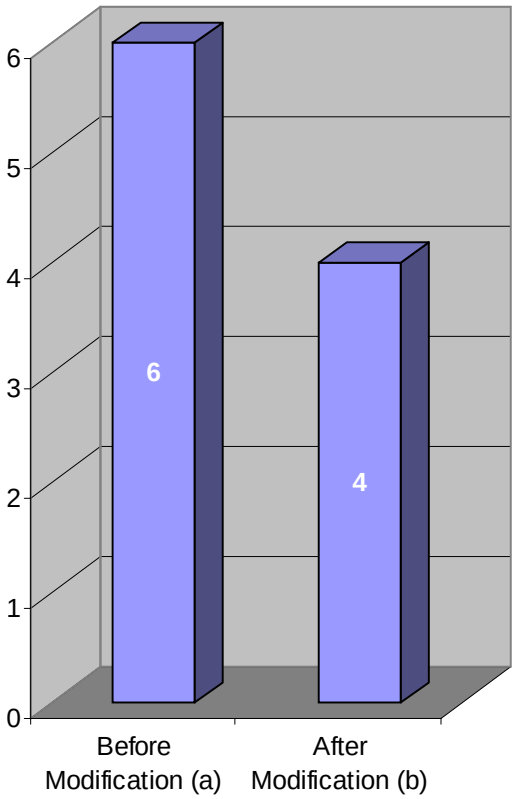

Figure 7-20: Number of memory cycles of one scan window access for the application example in figure 7-19 at page 169.

The Data Map Compression

Data map compression may be applied if the scan step or the distance between scan lines is larger than the scan window dimensions. In such a case data memory is wasted.

Figure 7-21a at page171 shows an application example, where the scan step is longer than the scan window width: the application data is segmented by columns of unused memory locations. The optimized application in figure 7-21b at page 171 is obtained after data map compression. This operation is performed through application of the transformation matrix *A* (see figure 7-21c at page171).

The general transformation matrix for data map compression is given in equation 7-12. The decision if data map compression is applied and the determination of c_1 and c_2 is currently a manual design step.

$$A = \begin{bmatrix} c_1 & 0 \\ 0 & c_1 \end{bmatrix} \qquad \textit{(Eq. 7-12)}$$

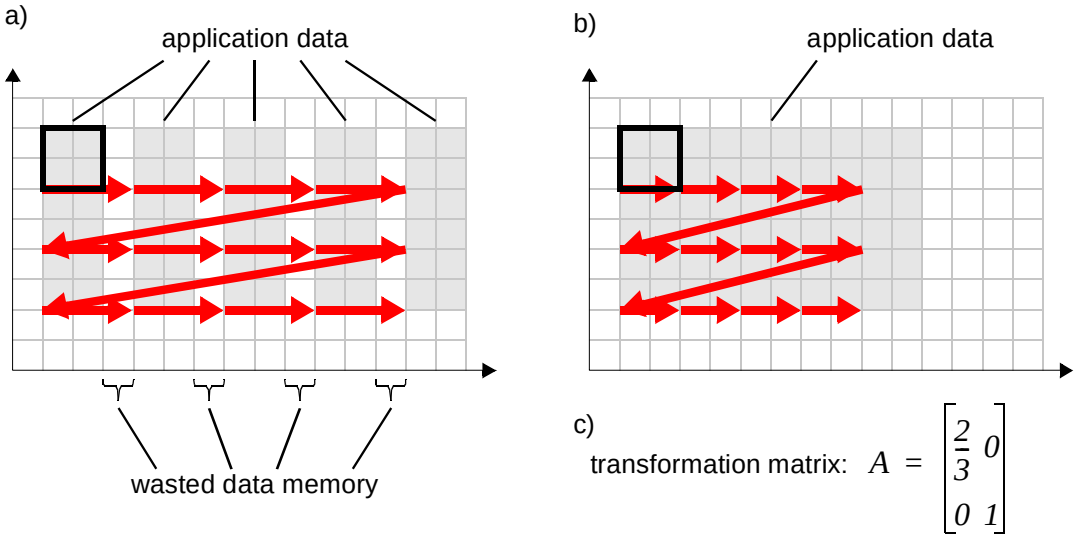

Figure 7-21: Data map compression example:
(a) before modification with wasted data memory, and
(b) after compression
(c) with transformation matrix *A*.

Equation 7-13 gives the required data memory in percent of the original application. Data map compression does not save memory cycles and hardware resources for the data sequencer design.

$$d = 100 \cdot c_1 \cdot c_1 \qquad \textit{(Eq. 7-13)}$$

The Data Map Transformation

The data map transformation may be performed to applications, where the scan steps are not parallel to one of the axis and no scan window overlapping occurs. Such applications have the highest demands on the data sequencer design (see section6.5.3 at page 133), have a poor potential for other optimizations, and waste data memory. The goal of the data map transformation is to rotate, stretch, and shear the application in a way, that the scan steps are parallel to one of the axis. The data map transformation does not reduce the number of memory cycles but decreases the amount of wasted memory and required data sequencer resources. Additionally it provides a fundamental basis for other access optimization techniques.

Figure 7-22a at page 172 shows a candidate application for data map transformation. The application data is mixed and surrounded by wasted memory locations. Loop-unrolling (see section 7.2 at page 162) would be of a poor result because the scan steps are diagonal in the data memory, i.e. burst operations can not be extended.

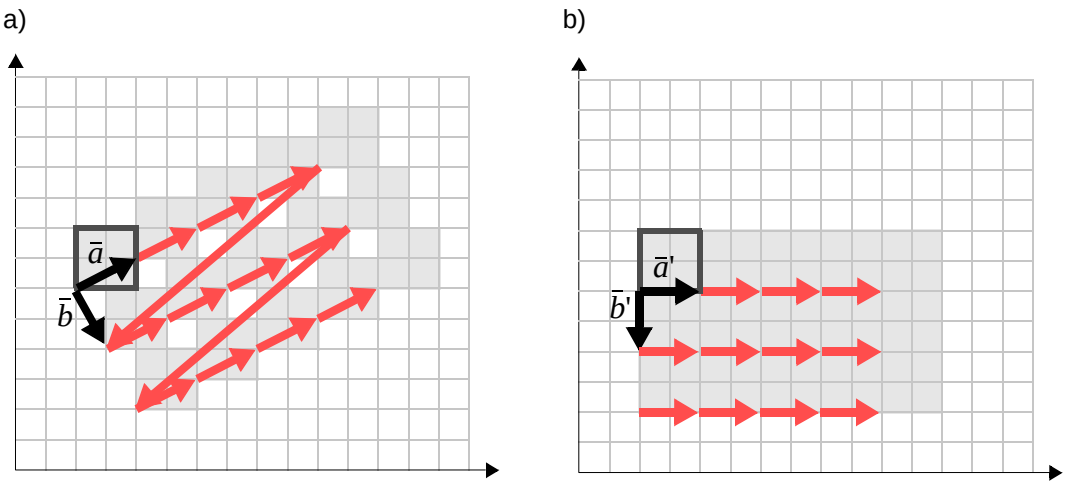

Figure 7-22: Application example:
(a) before data map transformation, and
(b) after modification.

Figure 7-22 pictures the application example before and after data map transformation. To perform the transformation the following vectors have to be determined:

$$\bar{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix} \qquad \textit{(Eq. 7-14)}$$

$$\bar{b} = \begin{bmatrix} b_x \\ b_y \end{bmatrix} \qquad \textit{(Eq. 7-15)}$$

$$\bar{a}' = \begin{bmatrix} a_x' \\ a_y' \end{bmatrix} \qquad \textit{(Eq. 7-16)}$$

$$\bar{b}' = \begin{bmatrix} b_x' \\ b_y' \end{bmatrix} \qquad \textit{(Eq. 7-17)}$$

While the vectors $\bar{a}$ and $\bar{b}$ are derived from the original application, the determination of $\bar{a}'$ and $\bar{b}'$ is a manual step. With equation 7-18 and equation 7-19 the universal transformation matrix has been derived (see equat i on7-20 at pa ge173).

$$\bar{a}' = A \cdot \bar{a} \qquad \textit{(Eq. 7-18)}$$

$$\bar{b}' = A \cdot \bar{b} \qquad \textit{(Eq. 7-19)}$$

$$A = \frac{1}{(a_x \cdot b_y) - (a_y \cdot b_x)} \cdot \begin{bmatrix} (a_x' \cdot b_y) - (a_y \cdot b_x') & (a_x \cdot b_x') - (a_x' \cdot b_x) \\ (a_y' \cdot b_y) - (a_y \cdot b_y') & (a_x \cdot b_y') - (a_y' \cdot b_x) \end{bmatrix} \quad \textit{(Eq. 7-20)}$$

For the application example in figure 7-22 at page 172 the vectors and the transformation matrix *A* are defined as follows:

$$\bar{a} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad \textit{(Eq. 7-21)}$$

$$\bar{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \textit{(Eq. 7-22)}$$

$$\bar{a}' = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \quad \textit{(Eq. 7-23)}$$

$$\bar{b}' = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \quad \textit{(Eq. 7-24)}$$

$$A = \begin{bmatrix} \frac{4}{3} & -\frac{2}{3} \\ -\frac{2}{3} & \frac{4}{3} \end{bmatrix} \quad \textit{(Eq. 7-25)}$$

	Before Transformation	**After Transformation**
memory cycles	no modification	
required data memory	120 addresses	60 addresses
data sequencer resources (see section 6.5.3 at page 133)	14 DPUs	6 DPUs

Table 7-4: Results of the data map transformation for the application example in figure 7-22 at page 172.

As summarized in table 7-4 the data map transformation of the application example halves the amount of required data memory. Further the data sequencer design to generate the modified scan pattern can also be optimized (see section 6.5.3 at page 133). This reduces the number of DPUs by 57% (see figur e7-23 at page174).

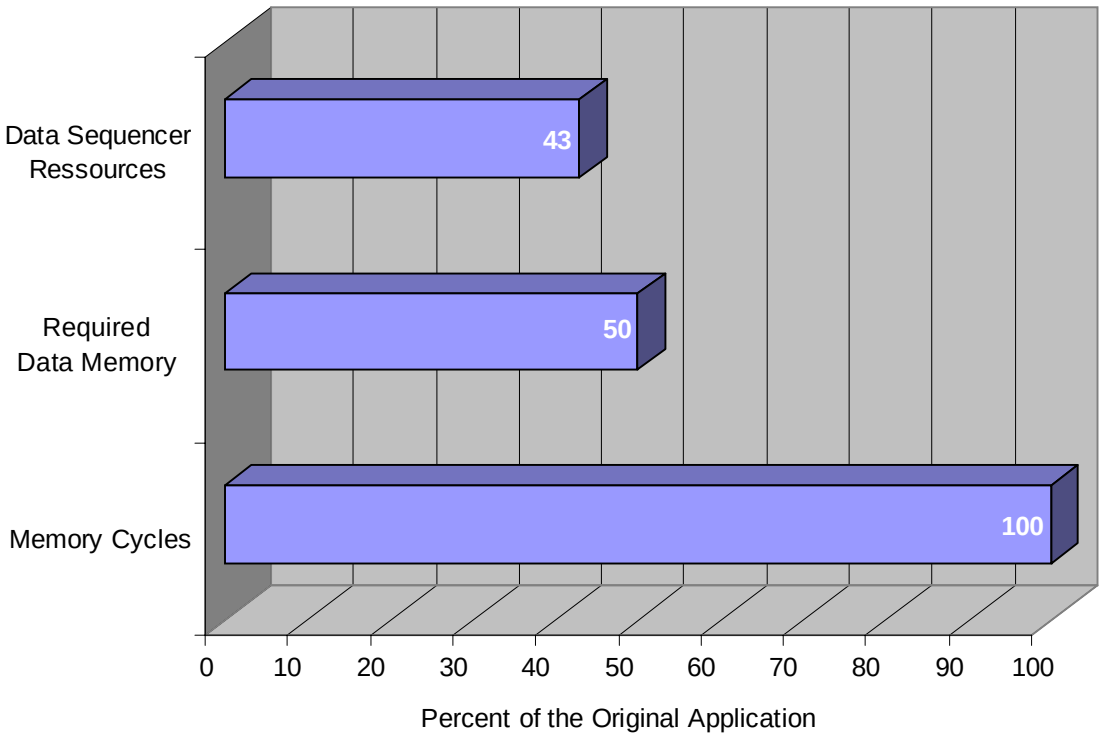

Figure 7-23: Required resources of the transformed example application of figure 7-22b at page 172 in percent of the original application (figure 7-22a at page172).

7.4 Scheduling of Memory Accesses

The sequence, in which data accesses are performed, is a determinant factor for the overall execution time of an application. Concerning the memory interface, there is an optimum scheduling for the memory accesses to achieve the best interface performance. This scheduling is called Optimum Data Access Scheduling. Regarding only the computational datapath (e.g. the KressArray, see section 2.7 at page 15) under consideration of the different execution times of single operators, there is also an optimum order for the data accesses to achieve the best reconfigurable datapath performance. This scheduling is called Optimum Data Computation Scheduling. Since the overall speed is limited by both, the memory interface and the reconfigurable datapath computation time, a trade-off between Data Access Scheduling and Data Computation Scheduling must be found. A description of the Data Computation Scheduling is given in [Kre96] or [Haa99].

Generally data accesses are grouped in read and write. Since computation data is needed before results can be written back, all read accesses are scheduled before the write accesses.

7.4.1 The Optimum Data Access Scheduling

The Optimum Data Access Scheduling gives at least one optimum order for the data accesses to achieve the highest memory interface speed of a given set of accesses. There are two possible memory access optimizations, which can be exploited by the Data Access Scheduling; burst operations (see section 7.1.4.3 at page 158) and parallel memory access (see section 7.1.4.1 at page 153). Both optimizations can be applied simultaneously.

Also the scan window overlap optimization improves the memory interface speed. But this optimization saves memory accesses completely and therefore it does not depend on an specific order of the accesses.

To determine the Optimum Data Access Scheduling a given scan window (e.g. see figure 7-9 at page 158) is examined. First the burst operations are determined. After that the memory accesses are assigned to the parallel memory banks.

Determination of the Burst Accesses

Figure 7-24 at page 176 presents an algorithm to find all burst accesses in a given scan window. Since burst operations work only in x-direction and start always on the lower x-address, each row of a scan window can be processed separately. The goal of the presented algorithm is to find only the longest burst accesses. A further constraint is, that there are no mixed burst operations, i.e. there are only read-, or write-bursts.

Determination of Concurrent Memory Accesses

Concurrent data accesses can only be performed on parallel memory banks. Depending on the number *n* of parallel banks the scan window rows are grouped. Having a number of *n* parallel memory banks, there are *n* sets A_n of burst accesses, each set assigned to one memory bank. If the number of scan window rows is higher than *n*, there are several scan window rows assigned to one memory bank (see figure7-3 at pa ge150).

The order, in which a set of burst accesses is performed on its assigned memory bank, also affects the execution time. Based on the physical memory organization several bursts may target the same bank of the DRAM [Bed98]. Since activating and precharging of memory banks costs additional time, these actions should be minimized. The best result is achieved, when all accesses of a scan window row are scheduled from left to right. Mixing the accesses of several scan window rows is not allowed. Further, if the application uses a small data map, maybe also two neighboring

```
begin
        burstnr=0
        while there are unprocessed scan window rows do
                while scan window row is not empty do
                        burstnr=burstnr+1
                        element=1
                        burst(burstnr,element)=next unprocessed element
                        while type of burst(burstnr,element)=
                                        type of next unprocessed element do
                                element=element+1
                                burst(burstnr,element)=next unprocessed element
                        endwhile
                endwhile
        enwhile
end
```

Figure 7-24: Algorithm to find the optimum burst accesses.

scan window rows are located in the same row of a memory bank (see addressing scheme in figure 7-10 at page 159). Therefore the scan window rows are processed from the bottom to the top of the scan window.

The Access Time of the Optimum Data Access Scheduling

This subsection deduces the access time to process a scan window, scheduled by the optimum data access scheduling.

Equation 7-26 calculates the exact time needed to perform one scan window access, based on the sets A_n of burst operations for all parallel memory banks *n*. It uses the function to calculate the execution time of a burst operation given in equation 7-7 at page 159.

$$t_{PDAS} = \left\lceil \forall_n \sum f_{tb}(A_n) \right\rceil \qquad \textit{(Eq. 7-26)}$$

Because of irregular delays like refresh cycles can not be considered, equation 7-27 at page 176 gives a sufficient approximation based on the overall number of memory accesses of a scan window (a) and the overall number of burst accesses (b). t_{acc} is the execution time of a single memory operation (apply address or access data), and n is the number of parallel memory banks. Here each memory operation (apply an address or access one data location) is counted.

$$t_{PDAS} \approx \left\lceil \frac{a+b}{n} \right\rceil \cdot t_{acc} \qquad \textit{(Eq. 7-27)}$$

7.4.2 Scheduling Trade-off

While the Optimum Data Access Scheduling gives the best utilization of all memory access techniques, the overall execution time may slow down because the order is disadvantageous for computations of the reconfigurable datapath. Since Data Access Scheduling and Data Computation Scheduling may be contrary there may be no good solution for a given application. This section introduces 2 scheduling methods to find a trade-off. Further hardware solutions are suggested to solve the scheduling problem.

Simple Scheduling

The Simple Scheduling calculates the Optimum Data Access Scheduling and the Optimum Data Computation Scheduling to find out if the application is bound by the data access time or by the computation time. Depending on this the Optimum Data Access Scheduling or the Optimum Data Computation Scheduling is applied.

This method gives good results, when an application is strongly dominated by data transfers or by computations. Otherwise this scheduling should not be applied.

Enhanced Scheduling

Also the Enhanced Scheduling is based on the Optimum Data Access Scheduling and the Optimum Data Computation Scheduling. All accesses of the Optimum Data Computation Scheduling are sorted according to the targeted memory bank. The accesses are then scheduled in the following way:

A distance d_n is calculated for all n bursts of the Optimum Data Access Scheduling giving the highest distance of two memory accesses of this burst in the sequential list of the Optimum Data Computation Scheduling.

Based on the distance d_n a weight w_n is calculated, which considers the burst length l_n (but not the order of the accesses):

$$w_n = \frac{d_n}{l_n} \qquad \textit{(Eq. 7-28)}$$

Figure 7-25 at page 178 presents the Enhanced Scheduling algorithm. It starts with the burst set of the Optimum Data Access Scheduling and splits all burst with *weight* > *maxweight*. *maxweight* is initially set to the overall number of data accesses. Therefore in the beginning no burst is split. The algorithm starts searching the minimum of execution time by decreasing the *maxweight* until *maxweight=1*. For *maxweight=1* all burst have the length $l_n=1$. Each time a burst set for a *maxweight* is computed its execution time is compared to the best burst set known at this point. The algorithm gives the best burst set for all values of *maxweight.*

```
begin
    maxweigth=number_of_data_accesses
    b1=split all bursts with weight>maxweigth
    t1=computation time of algorithm with b1
    t_result=t1
    b_result=b1
    while maxweigth>1 do
        maxweigth=maxweigth-1
        b1=split all bursts with weight>maxweigth
        t1=computation time of algorithm with b1
        if t1<t_result do
            t_result=t1
            b_result=b1
        endif
    endwhile
    b_result is the burst set of the Enhanced Scheduling
end
```

Figure 7-25: Algorithm to find the Enhanced Scheduling.

To specify the order of the bursts a second weight o_n is defined as the sum of the positions of all accesses of a burst in the Optimum Data Computation Scheduling list. The bursts of the Enhanced Scheduling are ordered with increasing o_n.

Optimum Scheduling with a 2-level Smart Interface

High performance applications on reconfigurable architectures are based on deep pipelined designs. To achieve an optimum data scheduling the smart interface is expanded to a 2-level smart interface. During operations the first smart interface level holds the data of the actual handle position. Data is sequenced from the data memory according to the Optimum Data Access Scheduling. The second smart interface level holds the data of the last handle position, where it is accessed by the computational datapath.

There are two conceivable solutions, which differ in the place, where the second smart interface level is located. If it can be integrated into the reconfigurable datapath, each register of the smart interface is set twice and each computation step they are switched. The reconfigurable datapath and the data sequencer use always different registers. In that case data can be sequenced with the Optimum Data Access Scheduling and all operators in the reconfigurable datapath have parallel access to their data.

If the second smart interface level can not be integrated into the reconfigurable datapath, an external scan cache is necessary (see figure 7-26 at page 179) and the data sequencer is supplemented with a second Scan Window Generator (SWG). Therefore this solution is more hardware prone. During operations the first SWG of the data

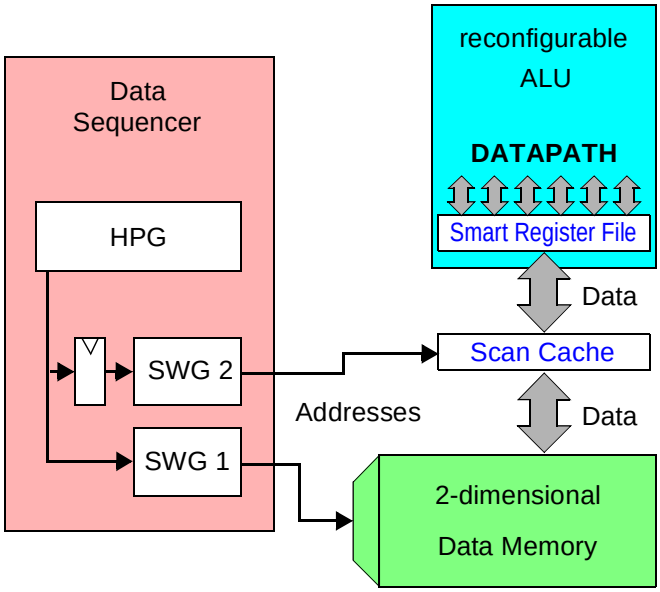

Figure 7-26: Supplemented hardware architecture, which supports optimum scheduling of data.

sequencer sequences data of the actual handle position from the data memory to the first smart interface stage (the scan cache) according to the Optimum Data Access Scheduling. The second SWG sequences the data of the last handle position according to the Optimum Data Computation Scheduling from the scan cache to the second smart interface stage, where it can be accessed in parallel by the computational datapath.

The disadvantage of both approaches is a certain hardware overhead, where the overhead of the second approach is higher, because the data sequencer has to be equipped with a second level of SWGs (one for each memory bank) in addition to the double size smart interface.

7.5 The Optimization Method

After the presentation of several optimization concepts, the proceeding of optimization has to be determined. Figure 7-27 at page 180 pictures the recommended order to apply the presented optimization techniques.

To obtain a good basis for further optimizations first a data map transformation is performed if needed. This basis is improved by the data map compression. Both techniques are high level storage scheme modifications (see section 7.3.2 at page 168).

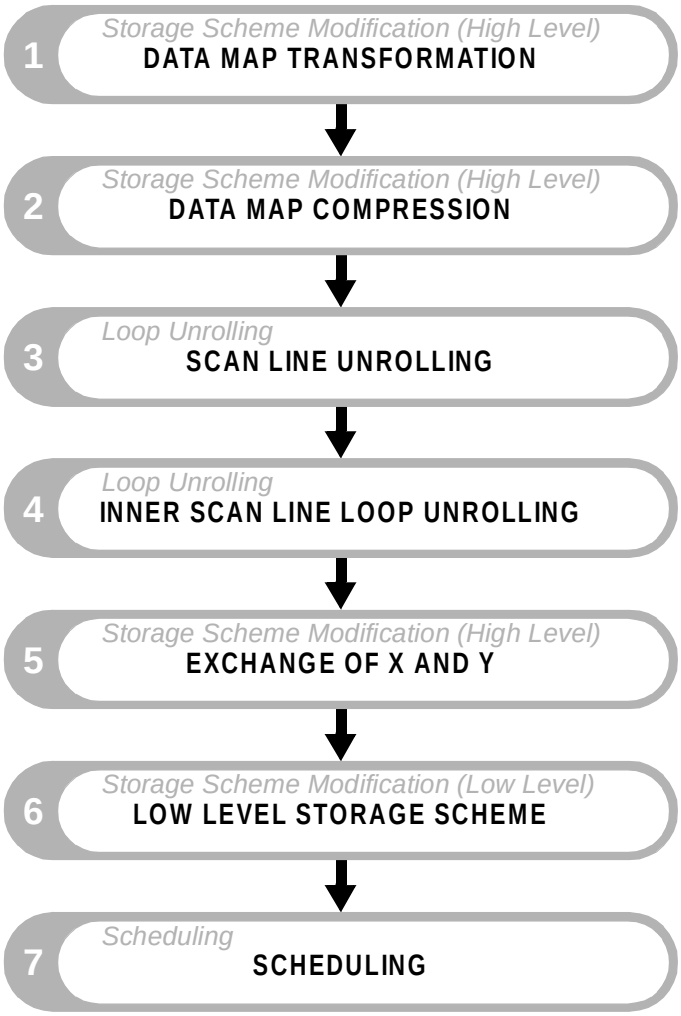

Figure 7-27: The optimization flow.

Next loop unrolling (section 7.2 at page 162) is performed. Because of the higher degree of optimization scan line unrolling is performed first (see section 7.2.2 at page 163). Then inner scan line loop unrolling is performed (see section 7.2.1 at page 162).

After loop unrolling the third high level storage scheme modification, exchange of x and y, is performed. This provides the basis for fine grain storage scheme optimization and the scheduling of the memory accesses. After the scheduling is generated also all hardware level optimizations are utilized.

7.6 Chapter Summary

This chapter has presented a novel two-dimensional memory organization, which supports parallel memory banks. On hardware level three access methods have been pointed out:

- concurrent access to parallel memory banks,
- burst access, and
- the scan window overlap optimization.

To achieve the optimum utilization of the hardware level access optimizations an algorithm has been described, which utilizes the following methods:

- scheduling of memory accesses,
- modification of storage schemes, and
- loop transformations.

The described memory access optimization method is based on the data sequencing concept and benefit from the flexibility of reconfigurable hardware. The effectiveness of the proposed memory access optimizations will be demonstrated with an image processing example in the next chapter.