Multimedia Rectangularly and Separably Addressable Memory

Georgi Kuzmanov Georgi Gaydadjiev Stamatis Vassiliadis

Computer Engineering Lab,

Electrical Engineering Dept., TU Delft, The Netherlands,

E-mail: {G.Kuzmanov, G.N.Gaydadjiev, S.Vassiliadis}@EWI.TUDelft.NL

Technical report CE-TR-2004-01

http://ce.et.tudelft.nl/publications.php

Abstract

In this report, we focus on the parallel access of randomly aligned rectangular blocks of visual data, common for various multimedia applications. As an alternative of traditional linearly addressable memories, we suggest a memory organization based on an $a \times b$ array of memory modules. A highly scalable data alignment scheme incorporating module assignment functions and a new generic addressing function are proposed. The addressing function implicitly embeds the module assignment functions and is separable, which potentially enables short critical paths and saves hardware resources. We also discuss the interface between the proposed memory organization and a linearly addressable memory accompanied with comprehensive examples. An implementation, suitable for MPEG-4 is presented and mapped onto an FPGA technology as a case study. Synthesis results indicate reasonably small hardware costs for an exemplary 512×1024 2D addressable space and a range of access pattern dimensions. The design is envisioned to be more cost-effective compared to related works. Regarding performance, our experiments suggest that a speedup of 8x can be expected.

I. Introduction

The problems of conflict-free parallel accesses of different data patterns out of a two-dimensional storage have been extensively explored for long time in several research areas. Vector processors designers have been interested in memory systems that are capable of delivering data at the demanding bandwidths of the increasing number of pipelines, see for example [1], [6], [8], [12]. Different approaches have been proposed for optimal alignment of data in multiple memory modules [1], [3], [8]–[10], [12]. Module assignment and addressing functions have been utilized in various interleaved memory organizations to improve the performance. In graphical display systems, researchers have been investigating efficient accesses of different data patterns: blocks (rectangles), horizontal and vertical lines, forward and backward diagonals [10], [11]. While all these patterns are of interest in general purpose vector machines and graphical display systems, rectangular blocks are the basic data structures in visual data compression. The most computationally intensive algorithms, like motion estimation and the discrete cosine transform, operate on square pixel blocks, requiring a huge data throughput. Therefore, the emerging visual data compression standards have narrowed the problems towards block (rectangularly) accessible memories with emphasis on high-performance implementations. Furthermore, to utilize the available bandwidth of a particular machine efficiently, new scalable memory organizations, capable of accessing rectangular pixel patterns are needed.

In this report, we propose an addressing function for rectangularly addressable systems, with the following characteristics:

- Rectangular sub-arrays out of a two-dimensional data storage can be accessed with high scalability. The addressing is separable, which saves addressing hardware. We also introduce implicit module assignment functions to further improve the designs. Finally, we propose a conflict free data routing circuitry avoiding large critical path penalties.
- Reasonably small hardware costs are shown by an FPGA case study implementation. In our experiments, we consider the maximum available on-chip memory of the Xilinx Virtex II Pro 2vp50ff1152 device, which is sufficient to implement a 512×1024-byte data storage. The proposed implementation requires from as little as 534 slices for 2×4-pixel patterns up to 3287 slices for 8×8 ones, which is between 1% and 13% of the today's reconfigurable device resources considered. Speedups around 8x are estimated for the case study FPGA implementations versus traditional linearly accessible memories.

The remainder of the report is organized as follows. Section II motivates the presented research and introduces the particular addressing problem. In Section III, the addressing scheme is described and the corresponding memory organization with a possible implementation are discussed. Case study synthesis results for FPGA technology are reported and related work is compared to our design in Section IV. Finally, the report is concluded with Section V.

II. MOTIVATION

In this section, we consider the memory addressing and accessing problem by considering the MPEG standards. The problems described here, as well as the solutions described later are, however, of a general nature regarding vector rectangular data accessing.

The addressing problem - a motivating example. Most of data processing in MPEG standards is not performed over separate pixels, but over certain regions (blocks of pixels) from a frame. This generates problems with data alignment and access in system memory. To illustrate these problems, let us consider the following *motivating example*. Assume a single port Linearly Addressable Memory (LAM) and a pixel plane divided into blocks with dimensions 4x2, with each pixel represented by a byte. Further, assume that the video information is stored as a scan-line (see Figure 1(a)) and that the system is capable of accessing 8 bytes per cycle. Obviously, neither of the blocks containing pixels {8, 9, 10, 11, 24, 25, 26, 27} and {26, 27, 28, 29, 42, 43, 44, 45} is accessible by a single memory transfer. This is because these blocks are not aligned into consecutive memory locations (see Figure 1(b)). Even though the memory system could be accessing all data, because it can access linearly 8 bytes in a single memory cycle, in fact it can access, for example, either bytes {26, 27, 28, 29} or bytes {42, 43, 44, 45}, but not all 8 {26, 27, 28, 29, 42, 43, 44, 45}. Consequently, even though an 8-byte memory bandwidth is available, redundant data fetches can not be avoided.

Another approach to process block-organized data may be to reorder data into the LAM. If we position blocks into consecutive bytes (Figure 1(c)), we will be able to access such blocks in a single memory cycle (e.g., pixels {8, 9, 10, 11, 24, 25, 26, 27}). In MPEG, however, some of the most demanding algorithms (e.g., motion estimation) require accessing block data at an arbitrary position in the frame, thus in memory. In the Figure 1(c) example, accessing block {26, 27, 28, 29, 42, 43, 44, 45} requires 4 cycles, even though the bandwidth is 8 bytes. This is because only two of its bytes can be accessed in one memory access cycle (i.e., either {26, 27}, or {28, 29}, or {42, 43}, or {44, 45}). Figure 1(c) suggests that in such cases data fetching may become even less effective than the scan-line alignment scheme.

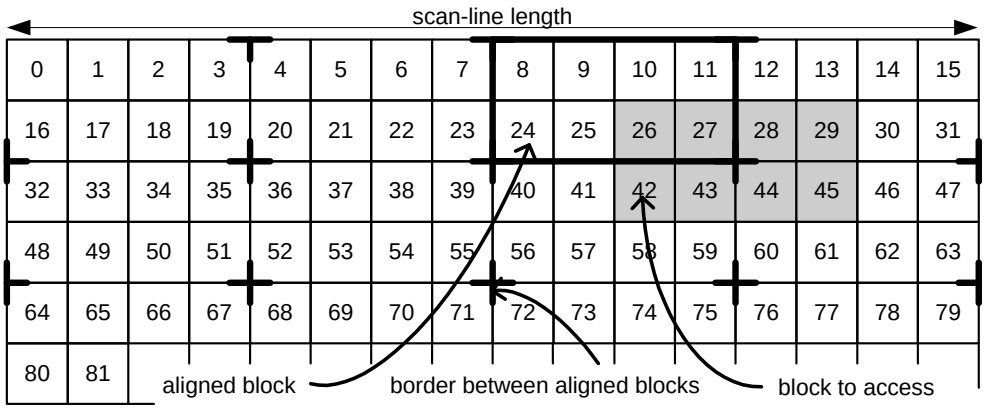
In the rest of the presentation, for conciseness, we will refer to blocks like {8, 9, 10, 11, 24, 25, 26, 27} in Figure 1(a) as aligned, and to the remaining blocks (like {26, 27, 28, 29, 42, 43, 44, 45}) as non-aligned. The borders between aligned blocks in the Figure are marked with thick line crosses.

Formal problem introduction and proposed solution. Let us assume a LAM with word length of *w* bits (*w* = 8, 16, 32, 64, 128) and the time for linear memory access to be T_{LAM}. The time to access a single $a \times b$ sub-array of 8-bit pixels, depending on its alignment in the LAM will be:

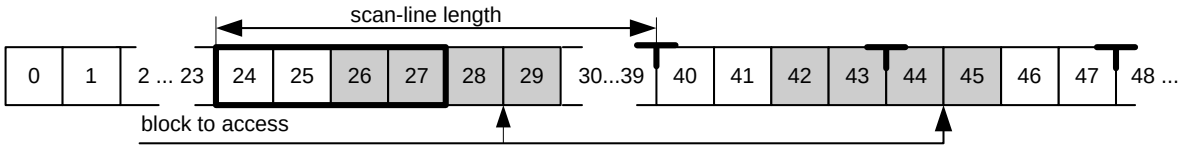
Aligned sub-array: $\frac{8 \cdot a \cdot b}{w} \cdot T_{LAM}$;

Not aligned sub-array: $(\frac{8 \cdot a}{w} + 1) \cdot b \cdot T_{LAM}$.

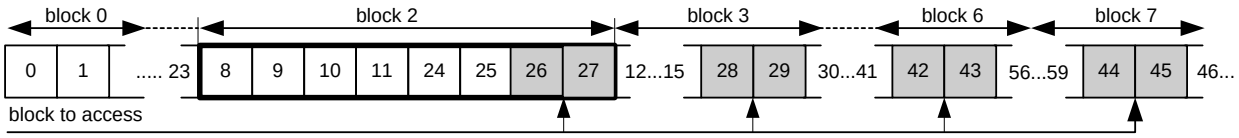
The time, required to access *N* $a \times b$ blocks will be:



(a) Pixels in a Video Frame;



(b) Scan-Line Alignment;



(c) Block-based Alignment.

Fig. 1. Addressing problem in LAM.

TABLE I

NUMBER OF LAM CYCLES IN DIFFERENT CASES

neither aligned	mixed	all aligned
$(\frac{8 \cdot n^2}{w} + n) \cdot N$	$(\frac{8 \cdot n^2}{w} + n - 1) \cdot N$	$\frac{8 \cdot n^2}{w} \cdot N$

All *N* blocks aligned: $N \cdot \frac{8 \cdot a \cdot b}{w} \cdot T_{LAM}$;

Neither of the blocks aligned: $N \cdot (\frac{8 \cdot a}{w} + 1) \cdot b \cdot T_{LAM}$;

Mixed: $N \cdot [\frac{1}{a} \cdot \frac{8 \cdot a}{w} + \frac{a-1}{a}(\frac{8 \cdot a}{w} + 1)] \cdot b \cdot T_{LAM} =$

$$= N \cdot (\frac{8 \cdot a}{w} + 1 - \frac{1}{a}) \cdot b \cdot T_{LAM} \quad (1)$$

By *mixed* access scenario we mean accessing both aligned and non-aligned blocks. In (1), we assume that the probability to access an aligned block is $\frac{1}{a}$, while for a non-aligned block it is $\frac{a-1}{a}$. For simplicity, but without losing generality, assume square blocks of $n \times n$, (i.e., *a*=*b*=*n*). Further assuming N blocks to access, we can estimate the number of LAM cycles as indicated in Table I. Obviously, the number of cycles to access an $n \times n$

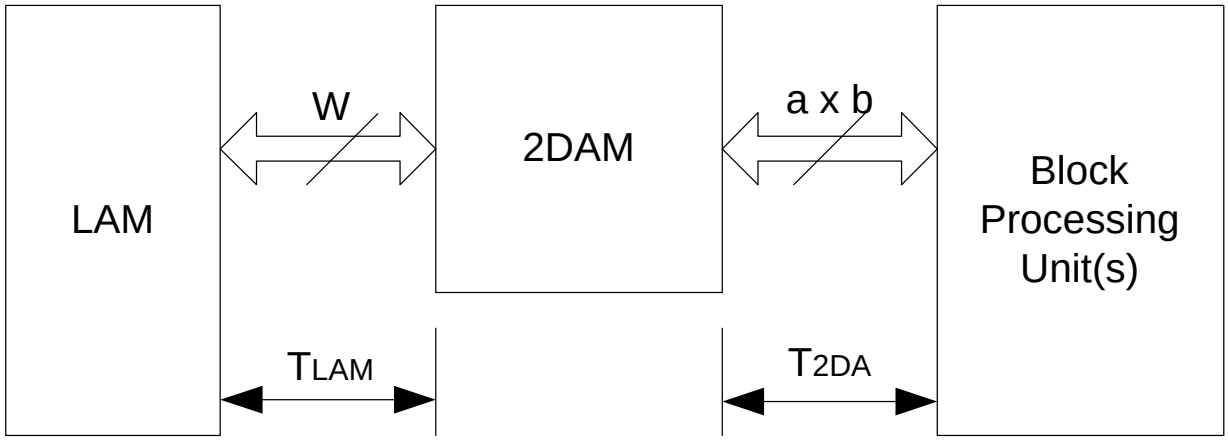


Fig. 2. Memory hierarchy with 2DAM.

TABLE II

ACCESS TIME PER $n \times n$ BLOCK IN LAM CYCLES. $t = \frac{T_{2DA}}{T_{LAM}}$.

n	w	LAM			2DAM	
		WC	Mix.	BC	Mix./BC	WC
8	8	72	71	64	8+t	64+t
	16	40	39	32	4+t	32+t
	32	24	23	16	2+t	16+t
16	8	272	271	256	32+t	256+t
	16	144	143	128	16+t	128+t
	32	80	79	64	8+t	64+t

block in a LAM is a square function of *n*, i.e., $O(n^2)$.

An appropriate memory organization may speed-up the data accesses. Consider the memory hierarchy in Figure 2 with time to access an entire $n \times n$ block from the 2D Accessible Memory (2DAM) to be T_{2DA}. In such a case, the time to access *N* $n \times n$ sub-blocks in the mixed access scenario will be:

$$\frac{N}{n} \cdot \frac{8 \cdot n^2}{w} \cdot T_{LAM} + N \cdot T_{2DA}, \quad [sec] \Leftrightarrow$$
$$(\frac{8 \cdot n}{w} + \frac{T_{2DA}}{T_{LAM}}) \cdot N, \text{ [LAM cycles].}$$

That is the sum of the time to access the appropriate number of aligned blocks ($\frac{N}{n}$) from LAM plus the time to access all *N* blocks from the 2DAM. It is evident that in a mixed access scenario, the number of cycles to access an $n \times n$ block in the hierarchy from Figure 2 is a linear function of *n*, i.e., *O(n)* and depends on the implementation of the 2D memory array. Table II presents access times per single $n \times n$ block. Time is reported in LAM cycles for some typical values of *n* and *w*. Three cases are assumed for LAM: 1.) neither of the *N* blocks is aligned - worst case (WC); 2.) mixed block alignment (Mix.); and 3.) all blocks are aligned - best case (BC). The last

two columns contain cycle estimations for the organization from Figure 2. In this case, both mixed and best case scenarios assume that aligned blocks are loaded from the LAM to the 2DAM first and then non-aligned blocks are accessed from the 2DAM. The 2DAM worst case (contrary to LAM) assumes that all blocks to be accessed are aligned. Even in this worst case, the 2DAM-enabled hierarchy may be better than LAM best case if the same aligned block should be accessed more than once. For example, assume accessing *k* times the same aligned block. In LAM, this would take $k \cdot \frac{8 \cdot n^2}{w} = [\frac{8 \cdot n^2}{w} + (k-1) \cdot \frac{8 \cdot n^2}{w}]$, while with 2DAM, it would cost $[\frac{8 \cdot n^2}{w} + (k-1) \cdot \frac{T_{2DA}}{T_{LAM}}]$ LAM cycles per block. Obviously, to have a 2DAM enabled memory hierarchy, faster than pure LAM, it would be enough if $\frac{8 \cdot n^2}{w} > \frac{T_{2DA}}{T_{LAM}}$. All estimations above strongly suggest that *a 2DAM with certain organization may dramatically reduce the number of accesses to the (main) LAM, thus considerably speeding-up related applications.*

III. Block Addressable Memory

In this Section, we present the proposed mechanism by describing its addressing scheme, the corresponding memory organization and a potential implementation.

Addressing Scheme. Assume $M \times N$ image data stored in $k = a \times b$ memory modules ($1 \leq a \leq M; 1 \leq b \leq N$). Furthermore, assume that each module is linearly addressable. We are interested in parallel, conflict-free access of $a \times b$ blocks at any (i,j) location, defined as:

$$B(i,j) = \{I(i+p, j+q) | 0 \leq p < a, 0 \leq q < b\},$$
$$0 \leq i \leq M-a, 0 \leq j \leq N-b.$$

To align data in *k* modules without data replication, we organize these modules in a two-dimensional $a \times b$ matrix. A module assignment function, which maps a piece of data with 2D coordinates *(i,j)* in memory module $(p,q) : 0 \leq p < a, 0 \leq q < b$, is required. We separate the function denoted as $m_{p,q}(i,j)$, into two mutually orthogonal assignment functions $m_p(i)$ and $m_q(j)$. We define the following module assignment functions for each module at position *(p,q)*:

$$m_p(i) = (i-p) \ mod \ a \tag{2}$$

$$m_q(j) = (j-q) \ mod \ b \tag{3}$$

The addressing function for module *(p,q)* with respect to coordinates *(i,j)* is defined as:

$$A_{p,q}(i,j) = (i \ div \ a + c_i) \cdot \frac{N}{b} + j \ div \ b + c_j \tag{4}$$

$$c_i = \begin{cases} 1, i \ mod \ a > p \\ 0, otherwise. \end{cases} \qquad c_j = \begin{cases} 1, j \ mod \ b > q \\ 0, otherwise. \end{cases}$$

Obviously, if $p = a-1 \Rightarrow c_i = 0$ for $\forall i$; if $q = b-1 \Rightarrow c_j = 0$ for $\forall j$, respectively. In essence, c_i and c_j are the module assignment functions, implicitly embedded into the linear address $A_{p,q}(i,j)$.

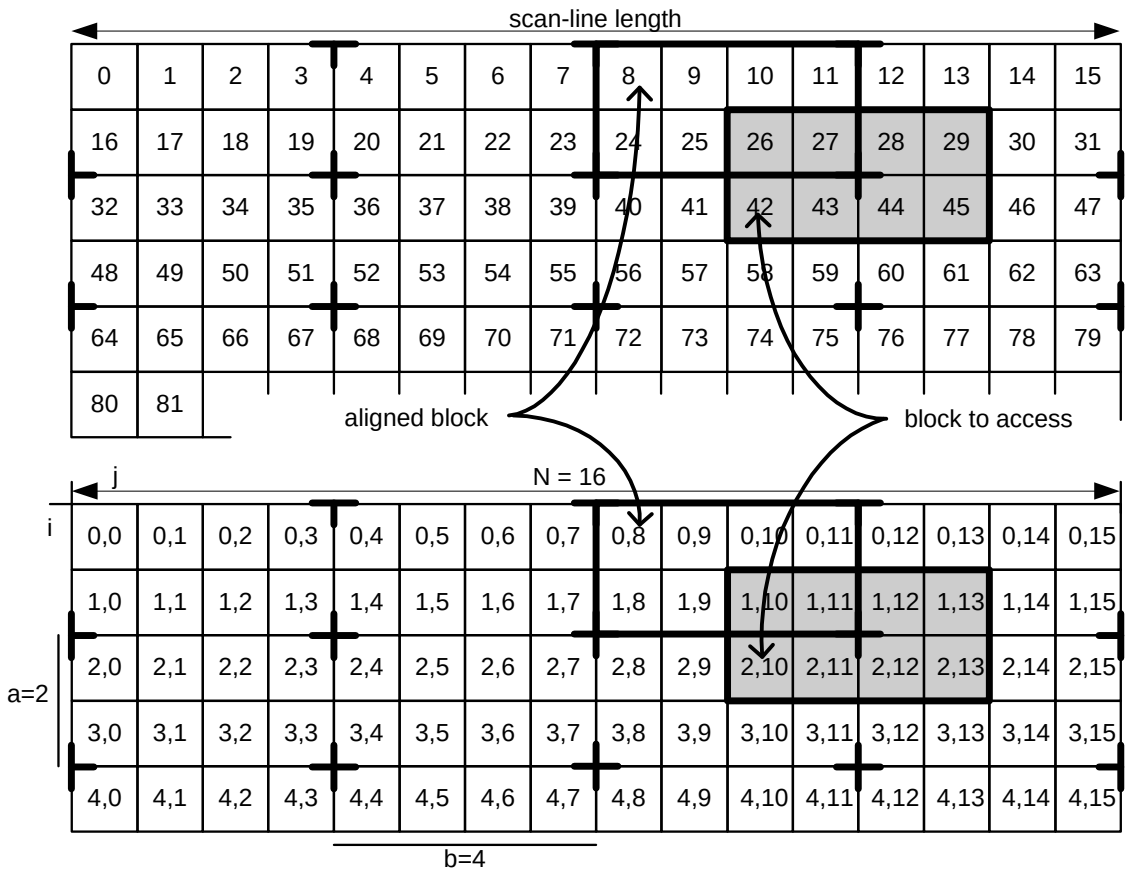


Fig. 3. Mapping of 2D organized pixels into a scan line sequence (considered example)

Example. Consider the motivating example of Section II and the pixel area from Figure 1(a). The same pixel area is mapped into a 2D addressing space with N=16 as depicted in Figure 3. In this new mapping, we address data by columns and rows, as 2D addressing is the actual addressing performed at algorithmic level. That is, byte 27 is referred to as *(1,11)*. Consequently, we have to perform the physical memory partitioning and assignment of data. Assume that data will be stored into linearly byte addressable memory modules, organized in a 2×4 matrix. Because in our example we have $5 \times 16 = 80$-byte memory, we subdivide the physical memory into 8 modules in total, 10 bytes each. Each pixel has to be allocated in a specific module by the assignment function. The memory module assignments of all pixels from the considered pixel area for a=2, b=4 are depicted in Figure 4(a). In the Figure, the pixel with 2D address *(1,11)* from Figure 3 is allocated by the module assignment function in module *(1,3)*. At the second addressing level, the linear address of each individual pixel within the module (*intra-module address*), has to be determined. The addressing function (4) generates a unique intra-module address within an uniquely assigned memory module, for each and every byte from the 2D addressing space. The intra-module address of pixel *(1,11)* determined by (4) is 2, denoted as A2 in Figure 4(b). Consequently, the proposed addressing scheme is in fact performed at two levels- module assignment and intra-module addressing.

As it has been stated, our scheme addresses and simultaneously accesses entire blocks rather than individual bytes. In the presented example, blocks are of dimension 2×4 bytes. By our definition, blocks are addressed by the 2D coordinates of their upper-left pixels. Consider the shaded non-aligned block $\{26\text{-}45\}$ from the motivating example. This block will be addressed as *B(1,10)*, see Figure 3. Note that the pixels of a block are accessed from all 8 modules simultaneously, in parallel. Using (2)-(4), we can calculate the linear address of the pixels from the considered block for each module *(p,q)* with respect to 2D address *i,j=(1,10)*:

- **module *(p,q)=(0,0)***

$$\left.\begin{array}{l} i \bmod a = 1 > p \Rightarrow c_i = 1 \\ j \bmod b = 2 > q \Rightarrow c_j = 1 \end{array}\right\} \Rightarrow A_{0,0}(1,10) = 7$$

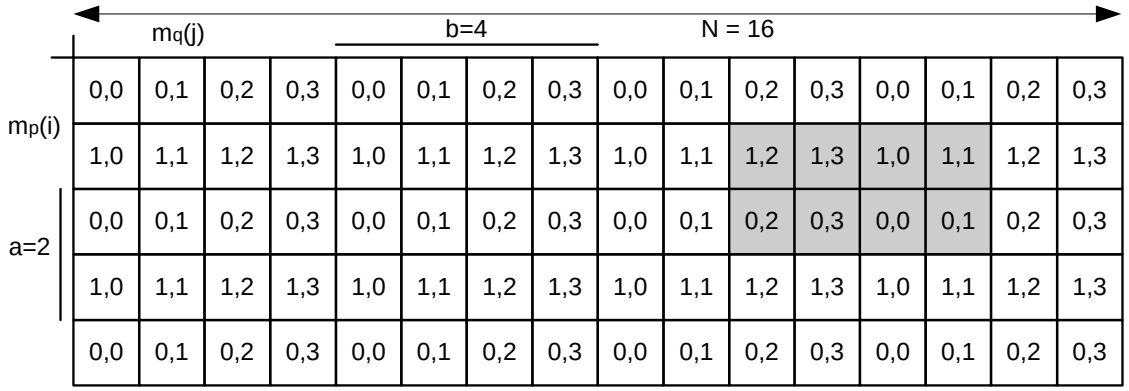
- **module *(p,q)=(1,3)***

$$\left.\begin{array}{l} i \bmod a = 1 = p \Rightarrow c_i = 0 \\ j \bmod b = 2 = q \Rightarrow c_j = 0 \end{array}\right\} \Rightarrow A_{1,3}(1,10) = 2$$

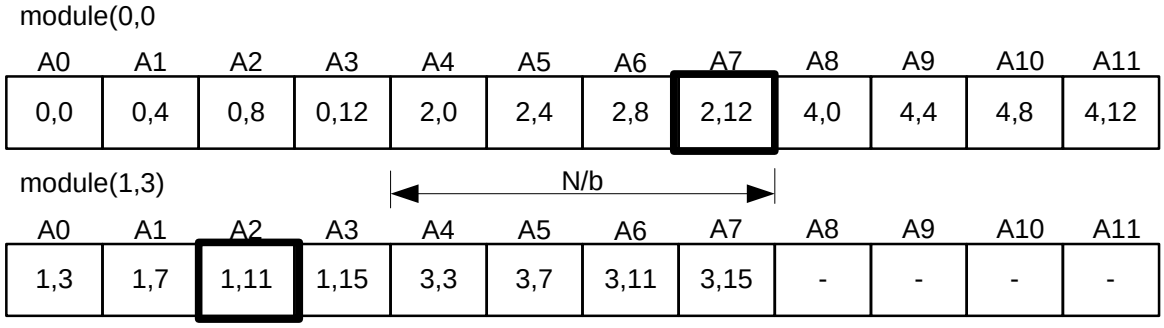
That is, the pixels of block *i,j=(1,10)* will be allocated at address 7 in module *(p,q)=(0,0)* and at address 2 in module *(p,q)=(1,3)*. Identically, the intra-module addresses of the remaining 6 pixels of the considered block can be calculated for each of the remaining 6 modules to be $A_{0,1}(1,10) = 7$, $A_{0,2}(1,10) = 6$, $A_{0,3}(1,10) = 6$, $A_{1,0}(1,10) = 3$, $A_{1,1}(1,10) = 3$, $A_{1,2}(1,10) = 2$. Figure 4(b) illustrates the internal linear addressing and data alignment within the considered two memory modules. Note that having the intra-module addresses of all pixels in the considered block, we only need to know which module contains the upper-left pixel *(i,j)=(1,10)* to reorder the data properly. The upper-left pixel of block *B(1,10)* is calculated (from the zeroes of (2) and (3)) to be located in module *(p,q)=(1,2). Thus, having each and every of the 8 block pixels localized in each and every of the 8 modules, we can access the entire block in one cycle by accessing all the modules in parallel.* Yet identically, it can be shown that any 2×4 block, regardless its position (thus including aligned blocks), can be accessed in a single cycle. Recall that block *B(1,10)* is the 2D notation of block $\{26\text{-}45\}$ from the motivating example. This block was accessible in 2 or 4 cycles from a conventional 8-byte LAM, thus 2 to 4 times slower than the proposed scheme at the same bandwidth of 8 bytes per cycle.

Memory Organization and Implementation. The key purpose of the proposed addressing scheme is to enable performance-effective memory implementations optimized for algorithms requiring the access of rectangular blocks. Designs with shortest critical paths are to be considered with the highest priority, as they dictate machine performance. Equations (2)-(4) are generally valid for any natural value of parameters *a*, *b* and *N*. To implement the proposed addressing and module assignment functions, however, we will consider practical values of these parameters. Since pixel blocks processed in MPEG algorithms have dimensions up to 16×16, values of practical significance for parameters *a* and *b* are the powers of two up to 16 (i.e., 1, 2, 4, 8, 16). For the particular implementation example we will consider the discussed block size - $a \times b = 2 \times 4$.

Module addressing. An important property of the proposed module addressing function is its *separability*. It



(a) Module assignments of the 2D pixel area



(b) 2D addresses and linear addressing within modules

Fig. 4. Memory modules assignment and internal addressing for *a=2, b=4, N=16.*

means that the function can be represented as a sum of two functions of a *single* and *unique* variable each (i.e., variables i and j). The separability of $A_{p,q}(i,j) = Ai_p(i) + Aj_q(j)$ allows the address generators to be implemented per column and per row (see Figure 5) instead of implemented as individual addressing circuits for each of the memory modules. Taking into account the separability of $A_{p,q}(i,j)$ and considering an arbitrary range of picture dimensions to be stored, we can define $C_h = N = 2^n, n \geq 4$ as "horizontal capacity" of the 2DAM (to be discussed later). The requirements for the frame sizes of all MPEG standards and for Video Object Planes (VOPs) [2] in MPEG-4 are constituted to be multiples of 16, thus, *N* is a multiple of 2^4 by definition. Assuming the discussed practical values of N and b, further analysis of Equation (4) suggests that $j \ div \ b + c_j < \frac{N}{b}$ and $(j \ div \ b + c_j)_{max} = \frac{N}{b} - 1$, i.e., no carry can be ever generated between $Ai_p(i)$ and $Aj_q(j)$. Therefore, we can implement $A_{p,q}(i,j)$ for every module *(p,q)* by simply routing signals to the corresponding address generation blocks without actually summating $Ai_p(i) + Aj_q(j)$. Figure 6(a) illustrates address generation circuitry of q-addresses ($Aj_q(j)$) for all modules except the first ($1 \leq q < b$). With respect to (4), if c_j is 1 the quotient *j div b* should be incremented by one, otherwise it should not be changed. To determine the value of c_j, a Look-Up-Table (LUT) with *j mod b* inputs can be used. For the assumed practical values of *a* and *b* (≤ 16), such a LUT would have at most 4 inputs, i.e., c_j *is a binary function of at most 4 binary digits*. Row p-addresses are generated identically. For *p=1* or *q=3*, $c_i = 0$, $c_j = 0$

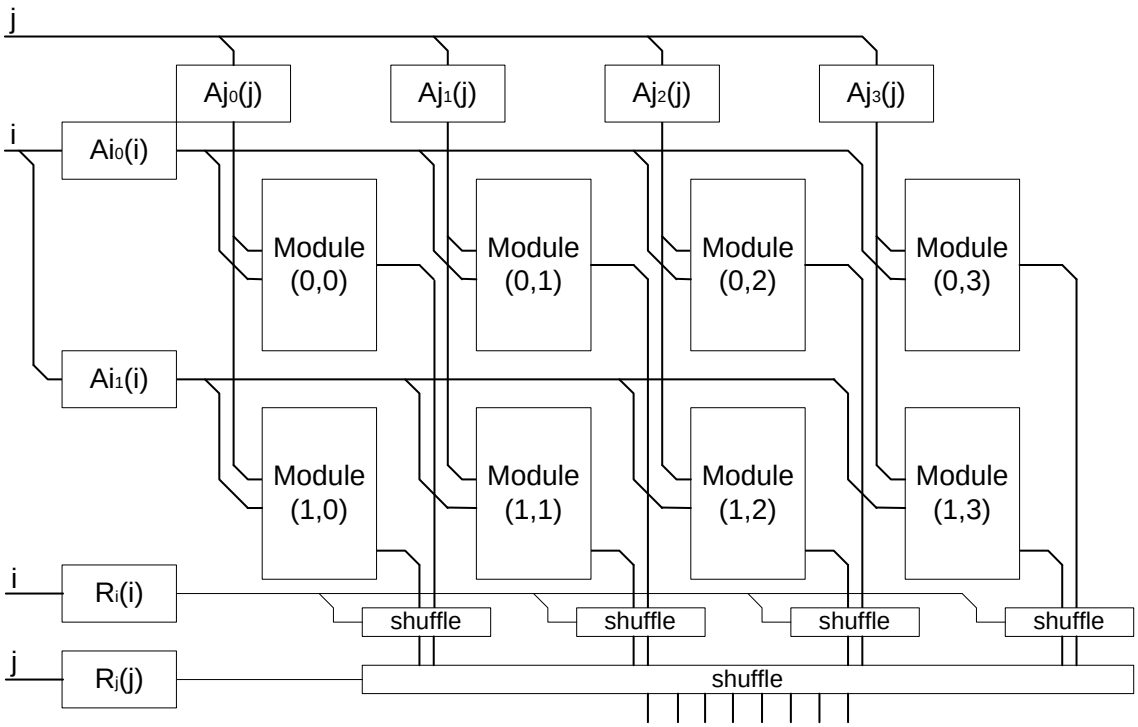
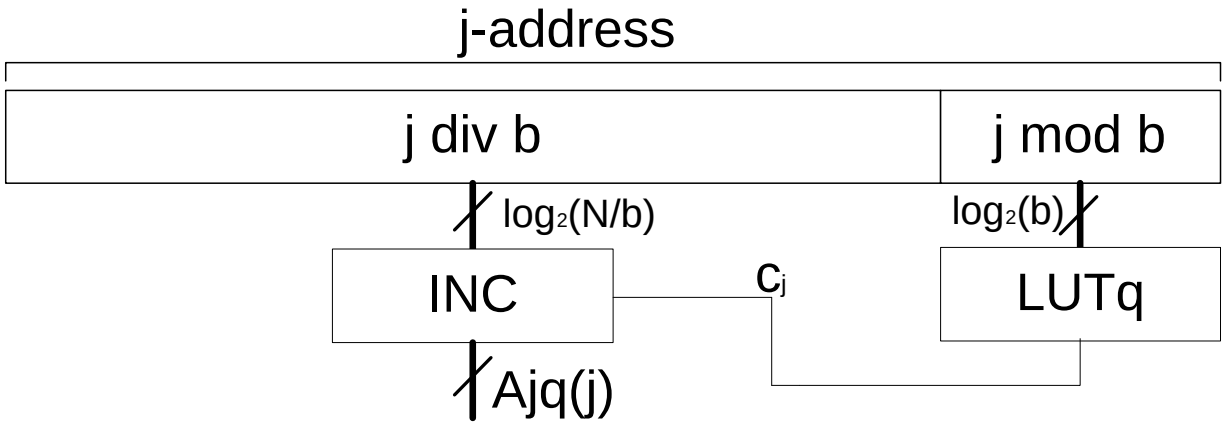


Fig. 5. 2DAM for *a=2*, *b=4* and $N = 2^n \geq 16$

respectively. Therefore, address generation in these cases does not require a LUT and an incrementor. Instead, it is just routing *i div a* and *j div b* to the corresponding memory ports, i.e., blocks $Ai_1(i)$ and $Aj_3(j)$ in Figure 5 are empty. Figure 6(b) depicts all 4 LUTs for the case $a \times b = 2 \times 4$. The usage of LUTs to determine c_i and c_j is not mandatory, fast pure logic can be utilized instead. However, we use LUTs for two main purposes: 1.) to illustrate the design concept; and 2.) LUTs are envisioned to fit better in the FPGA implementation considered further in this report.

Data routing circuitry. In Figure 5, the shuffle blocks, together with blocks $R_p(i)$ and $R_q(j)$, illustrate the data routing circuitry. The shuffle blocks are in essence circular barrel shifters, i.e. having the complexity of a network of multiplexors. An $n \times n$ shuffle is actually an $n \rightarrow 1$ n-way multiplexor. In the example from Figure 5, the *i*-level shuffle blocks are four ($2 \rightarrow 1$) 16-bit multiplexors and the *j*-level one is ($4 \rightarrow 1$) 64-bit. To control the shuffle blocks, we can use the module assignment functions for $p = q = 0$, i.e., $R_i(i) = i \ mod \ a$ and $R_j(j) = j \ mod \ b$. These functions calculate the *(p,q)*-coordinates of the "upper-left" pixel of the desired block, i.e., pixel *(i,j)*. For the assumed practical values of *a* and *b* being powers of two, the implementation of $R_i(i)$ and $R_j(j)$ is simple routing of the least-significant $log_2(a)$ -bits (resp. $log_2(b)$) to the corresponding shuffle level.

2DAM capacity. Earlier, we have defined the "horizontal capacity" of 2DAM as $C_h = N = 2^n, n \geq 4$. C_h is the *maximal scanline length in bytes (pixels), the 2DAM can store without addressing conflicts*. The "vertical capacity" of 2DAM is denoted as C_v and defined as the *maximal number of C_h-byte (C_h-pixel) scanlines the 2DAM can*



(a) Generation Circuit of q-addresses for $1 \leq q < b$

j mod b		c_j			i mod a	c_i p=0
		q=0	q=1	q=2		
0	0	0	0	0	0	0
0	1	1	0	0	1	1
1	0	1	1	0	-	-
1	1	1	1	1	-	-

(b) LUTs contents for *a=2, b=4*

Fig. 6. Module address generation

store. Finally, *the capacity C_{2DAM} of a 2DM is defined as the couple ($C_h \times C_v$)-bytes (pixels)*, rather than as a single number of bytes.

LAM Interface. Figure 7 depicts the organization of the interface between LAM and 2DAM (recall Figure 2) for the modules considered in Figure 5. Data bus width of the LAM is denoted by W (in number of bytes). In the particular example, W is assumed to be 2, therefore modules have coupled data busses. For each *(i,j)* address, the AGEN block sequentially generates addresses to the LAM and distributes write enable (WE) signals to a corresponding module couple. Two module WE signals (WE_i, WE_j) are assumed for easier row and column selection. In the general case, the AGEN block should sequentially generate $\frac{a \cdot b}{W}$ LAM addresses for each *(i,j)* address. Provided that pixel data is stored into LAM in scan-line manner, the set of LAM addresses to be generated is defined as follows:

$$A_{LAM}(i,j) = \{a \cdot (i \ div \ a) + k\} \cdot N + b \cdot (j \ div \ b) + l \cdot W$$

Which, assuming that only aligned blocks will be accessed from the LAM (i.e., *(i,j)* are aligned), can be simplified:

$$A_{LAM}(i,j) = (i + k) \cdot N + j + l \cdot W \tag{5}$$

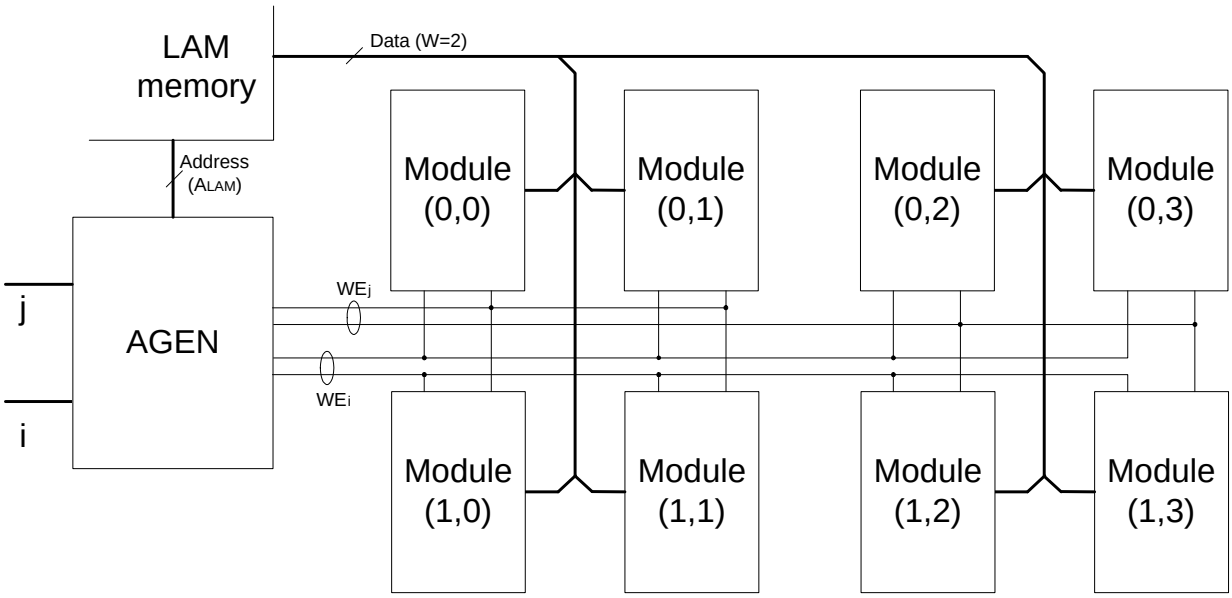


Fig. 7. LAM interface for *W=2, a=2, b=4*

$$k = 0, 1, ..., a-1; l = 0, 1, ..., \frac{b}{W} - 1.$$

In the 2DAM, the data words should be simultaneously written in modules:

$$(p, q) = (k, l \cdot W), (k, l \cdot W + 1), ..., (k, l \cdot W + W - 1) \tag{6}$$

at local module address:

$$A_{p,q}^{LAM}(i, j) = (i \ div \ a) \cdot \frac{N}{b} + j \ div \ b. \tag{7}$$

Note, that accessing only aligned blocks from the LAM enables thorough bandwidth utilization. When only aligned blocks are addressed, all address generators issue the same address, due to (4). Therefore, during write operations into 2DAM, the same addressing circuitry can be used as for reading. If the modules are true dual port, the write port addressing can be simplified to just proper wiring of both *i* and *j* address lines because the incrementor and the LUTs from Figure 6(a) are not required. Therefore, module addressing circuitry is not depicted in Figure 7.

Addressing consistency. In the following, we will prove that the described scheme provides a consistent LAM and 2DAM addressing. It means that each and every byte is allocated in the same memory module and at the same intra-module address by both LAM and 2DAM addressing schemes.

Lemma 1 $x \ mod \ z = x - n \cdot z \ \ iff \ 0 \leq x - n \cdot z < z; \ \forall x, n, z \in N.$

Proof. 1. If $x \ mod \ z = x - n \cdot z \ \ \Rightarrow \ 0 \leq x - n \cdot z < z; \ \forall x, n, z \in N$ is true by the definition of *mod* operation. 2. If $\ 0 \leq x - n \cdot z < z \Rightarrow x \ mod \ z = x - n \cdot z; \ \forall x, n, z \in N$. Let $x \ mod \ z = x - p \cdot z$. Then, by definition $0 \leq x - p \cdot z \leq z$. Assume $p \neq n \Rightarrow |p - n| \geq 1$. We derive the system:

$$\left\| \begin{array}{l} 0 \le x - n \cdot z < z \\ 0 \le x - p \cdot z < z \end{array} \right.$$

Its only solution *p* = *n* contradicts to the assumption ■

Lemma 2 $(x-y)\ mod\ z = (x\ mod\ z - y)\ mod\ z; \forall y < z; \forall x, y, z \in N$

Proof. By definition $x\ mod\ z = x - n1 \cdot z$ and $(x\ mod\ z - y)\ mod\ z = (x\ mod\ z - y) - n2 \cdot z.$ ⇒ By substitution and based on Lemma 1, we derive: $(x\ mod\ z - y)\ mod\ z = (x - n1 \cdot z - y) - n2 \cdot z = (x-y) - (n1+n2) \cdot z = (x-y)\ mod\ z$ ■

Lemma 3 $(x\ div\ y) \cdot\ y = x - x\ mod\ y$

Proof. $\left\| \begin{array}{l} x\ mod\ y = p \\ x\ div\ y = k \\ k \cdot y + p = x \end{array} \right.$ $\Rightarrow$ $\begin{array}{l} (x\ div\ y) \cdot y = \\ = k \cdot y = x - p = \\ = x - x\ mod\ y \end{array}$ ■

Theorem 1 (Consistency between the 2DAM and the LAM addressing schemes). *Assume the 2DAM and LAM addressing interface schemes defined by (2)-(4) and (5)-(7), respectively. Any byte* $(i^{`}, j^{`})$ *is allocated in the same memory module at the same intra-module address by both addressing schemes.*

Proof. Consistency of module assignments. Consider byte $(i^{`}, j^{`})$. In consistence with (5), we define $k = i^{`}\ mod\ a$ and $l = (j^{`}\ mod\ b)\ div\ W$. Considering the LAM interface and Lemma 3, the module, where byte $(i^{`}, j^{`})$ should be stored is calculated as follows:

$$(p,q) = (k, l \cdot W + (j^{`}\ mod\ b)\ mod\ W) =$$
$$= (k, \{(j^{`}\ mod\ b)\ div\ W\} \cdot W + (j^{`}\ mod\ b)\ mod\ W) =$$
$$= (k, (j^{`} mod\ b) - (j^{`} mod\ b) mod W + (j^{`} mod\ b) mod W)$$

$$\Rightarrow (p,q) = (k, j^{`}\ mod\ b) \tag{8}$$

Considering (2)-(3) for the 2DAM module allocation and Lemma 2, we derive:

$m_p(i^{`}) =$	$m_q(j^{`}) =$
$= (i^{`} - p)\ mod\ a = 0$	$= (j^{`} - q)\ mod\ b = 0$
$(i^{`} mod\ a - p)\ mod\ a = 0$	$(j^{`} mod\ b - q)\ mod\ b = 0;$
$(k-p) mod\ a = 0; k < a$	$j^{`} mod\ b\ < b$

$$\Rightarrow p = k;\ q = j^{`} mod\ b \tag{9}$$

Equations (8) and (9) indicate that any byte $(i^{`}, j^{`})$ will be allocated in the same memory module both by the LAM interface and by the 2DAM read circuitry.

Consistency of intra-module addresses. Assume ***(i,j)*** is the aligned block, containing byte $(i^{'}, j^{'})$, i.e., $i \ div \ a = i^{'} \ div \ a$, $j \ div \ b = j^{'} \ div \ b$. Consider (4):

$A_{p,q}(i^{'}, j^{'}) = (i^{'} \ div \ a + c_i) \cdot \frac{N}{b} + j^{'} \ div \ b + c_j$, from (9): $p = i^{'} \ mod \ a$ and $q = j^{'} \ mod \ b \Rightarrow c_i = c_j = 0, \ \Rightarrow$ $A_{p,q}(i^{'}, j^{'}) = (i \ div \ a) \cdot \frac{N}{b} + j \ div \ b$, identical to (7) ■

Example. We consider a single (arbitrary chosen) byte and show that it is allocated in the same memory module and at the same intra-module address both by the LAM and by the 2DAM addressing schemes.

Assume that visual data is scan-line aligned in LAM with word length of 2 bytes and big-endian convention. Consider the byte with 2D address *(1,11)*, see Figure 3. The memory hierarchy of Figure 2 indicates that byte *(1,11)* has to be loaded from the LAM into the 2DAM by means of the proposed LAM interface. Assuming that the 2DAM is first loaded in its entirety, all aligned blocks of the considered 5×16-byte area are to be loaded from the LAM into the 2DAM. Byte *(1,11)* is assigned in the LAM as part of aligned block *(0,8)*. The LAM addresses of the four 2-byte words containing the pixels of the block are $A_{LAM} = 8, 10, 24, 26$, see Figure 3. The LAM address of the 2-byte word, containing the considered pixel *(1,11)* is calculated from (5) to be: $A_{LAM}(0,8)_{k=1,l=1} = (0{+}1){\cdot}16{+}8{+}1{\cdot}2 = 26$. Recall Figure 3, where byte *(1,11)* had LAM address 27. Thus, in the assumed big-endian LAM convention, the considered byte 27 is the most significant byte of the 2-byte memory word aligned at address 26. Considering (6), this 2-byte word should be stored into modules *(1,2)* and *(1,3)*, see Figure 7. The most significant byte, i.e., byte 27, should be stored into module $(p,q)_{k=1,l=1} = (k, l \cdot W + W - 1) = (1,3)$. Its intra-module address with respect to the LAM interface is calculated from (7) to be:

$$A_{1,3}^{LAM}(0,8) = (0 \ div \ 2) \cdot \frac{16}{4} + 8 \ div \ 4 = 2$$

That is, *byte* (1,11) *with LAM address 27, will be stored by the LAM-to-2DAM interface into module (1,3) at intra-module address 2.*

Consider the 2DAM addressing scheme, the shaded non-aligned block *(1,10)* in Figure 3 and Figure 4, and (2)-(4). Indeed, *considering the 2DAM addressing scheme, byte* (1,11) *can be read from address location 2 of module* (1,3), as it was shown in the previous example.

Critical paths. Regarding the performance of the proposed design, we should consider the created critical path penalty. Assuming generic synchronous memories where addresses are generated in one cycle and data are available in another, we separate the critical paths into two: address generation and data routing. For the proposed circuit implementation, the address generation critical paths (CP_A) is determined by:

$$CP_A = max(CP_{add\frac{M}{a}}, CP_{add\frac{N}{b}}) + CP_{LUT}.$$

That is the critical path of either a $log_2(\frac{M}{a})$-bit or a $log_2(\frac{N}{b})$-bit adder, whichever is longer, and the critical path of one (max. 4-input) LUT. The data routing critical path (CP_D) is:

$$CP_D = CP_{mux_a} + CP_{mux_b}.$$

That is, the sum of the critical paths of one $a \rightarrow 1$ multiplexor and one $b \rightarrow 1$ multiplexor.

IV. Experimental Results and Related Work

In the previous sections, we have considered the theoretical aspects of our proposal illustrated by simplified examples. In this section, an experimental case study for a number of real-world FPGA-based designs is presented, followed by a comparison to other related works reported in literature.

Case study: A generic VHDL model of the memory organization has been developed and synthesized for the recent Virtex II Pro FPGA technology of Xilinx. We assume reconfigurable technology for two reasons. First, showing the viability of the organization in reconfigurable technology also proves its viability to all other current and near future technologies. Second, we envision, for cost-efficiency, that assuming MPEG specific requirements, the organization may be incorporated in a reconfigurable augmented processor [13]. Table III contains synthesis results for the 2vp50ff1152 FPGA device (the last column displays some of the resources available on the chip). The on-chip memory volume allows frames or VOPs sized up-to *512x1024* pixels to be stored. It should be noted that more than one frame can be stored in the memory and accessed, depending on the particular frame format. For example, up-to fourteen CIF frames (*144x176*) can be stored into the implemented *512x1024* storage. This issue is much more beneficial in MPEG-4, where the arbitrary shaped VOPs to be stored vary both in size and number for each particular codec session. Synthesis data for practical MPEG pattern sizes of *2x4, 4x8, 8x8 and 16x16-pixels* indicate that respective structures can be efficiently implemented with a fraction of the available FPGA resources. Only the *16x16* pattern creates a resource conflict with regard to the available IO pins of the chip. This conflict, however, should not be considered as a problem, since structures with bandwidth of that magnitude are usually intended for on-chip implementations. In the 'Adders' rows of Table III, the notation 'bits/#' denotes the number of bits in an adder and the corresponding number of such adders, respectively. Results indicate that in the most common case of 8×8 block patterns, 3287 Virtex II Pro slices are required, which is 13% of the today's 2vp50ff1152 FPGA device resources.

In Table IV, transfer speedup estimations are presented, assuming $T_{LAM} = 10ns$. Calculations are made according to the figures and notations presented in Table II. In BC, all blocks are assumed to be non-aligned, while in WC the very unlikely scenario that all blocks are aligned and accessed only once is considered. T_{2DA} values are derived from the synthesis reports for the designs considered in Table III. Figures in Table IV indicate that even in the unfavorable case when 2DAM is slower than the LAM, considerable transfer speedups of up to 8x can be achieved, due to the proposed memory organization.

Related work. Accessing blocks of memory has been in the hearts of vector (array) processors researchers and developers for long time. Two major groups of memory organizations for parallel data access have been reported

TABLE III

SYNTHESIS FOR FRAMES UP-TO 512X1024 (DEVICE 2VP50FF1152).

$a \times b$	2 x 4	4 x 8	8 x 8	16 x 16	Avail.
2-1mux	192	1280	3072	16384	N.A.
Adders:	4	10	14	30	N.A.
bits/#	8/1	7/3	6/7	5/15	N.A.
bits/#	8/3	7/7	7/7	6/15	N.A.
# Slices	534	1512	3287	15408	24640
%	1	6	13	63	100
# LUT4	928	2630	5723	26805	49280
%	1	5	11	54	100
IOs	100	292	548	2084	756
BRAM	8x	32x	64x	256x	522K
	64K	16K	8K	2K	

TABLE IV

ESTIMATED TRANSFER SPEEDUPS FOR $T_{LAM} = 10ns$

$a \times b$	T_{2DA}	t= $\frac{T_{2DA}}{T_{LAM}}$	w	Transfer Speedup		
				BC	Mix.	WC
8x8	16,7ns	1,67	8	7,45	7,34	0,97
			16	7,05	6,88	0,95
			32	6,54	6,27	0,91
16x16	18,8ns	1,88	8	8,03	8,00	0,99
			16	8,05	8,00	0,99
			32	8,10	8,00	0,97

in literature - organizations with and without data replication (redundancy). We are interested only in those without data replication. Another division is made with respect to the number of memory modules - equal to the number of accessed data points and exceeding this number. Organizations with a prime number of memory modules can be considered as a subset of the latter. An essential implementation drawback of such organizations is that their addressing functions are non-separable and more complex, thus slower and costly to implement. We have organized our comparison with respect to block accesses, discarding other data patterns, due to the specific requirements of visual data compression. It should be noted, however, that our design can be easily augmented to accesses horizontal and vertical $a \times b$ lines, just by slightly modifying the module assignment functions and preserving the same addressing function.

To compare designs, two basic criteria have been established: scalability and implementation drawbacks in terms of speed and/or complexity. Comparison results are reported in Table V. Budnik and Kuck [1] described a scheme

TABLE V

COMPARISON TO OTHER PROPOSED SCHEMES

Related Work	scalability	# modules	implementation drawbacks or limitations
Budnik, Kuck [1]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	prime $m > N = 2^n$	*mod(m)*, crossbar, no addressing
Lawrie [8]	$\sqrt{N} \times \sqrt{N}$	$m = 2.N; N = 2^{2n+1}$	*mod(m)*, no addressing
Voorhis, Morin [12]	$p \times q$ from $M \times N$	$m \geq p \times q$	not separable,*mod(pq)*,*mod(pq+1)*,
Kim, Prasanna [3]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	*m = N*	certain blocks are inaccessible
De-lei Lee [9]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	*m = N*	many modules for higher *N*
Sproull et al. [11]	8×8	8×8	time-space multiplexing, not general
Park [10]	$p \times q$ from $M \times N$	prime $m > p \times q$	not separable, many adders, big LUTs
HiPAR-DSP [5], [14]	$N \times N$	$m = (1 + N)^2$	$2 \times N + 1$ additional modules, *mod(m)*
HiPAR-DSP16 [4]	$p \times q$ from $M \times N$	$m >> p \times q$	big number of modules, *mod(m)*
This proposal	$p \times q$ from $M \times N$	$m = p \times q$	none of the above, rectangular patterns only

for conflict free access of $\sqrt{N} \times \sqrt{N}$ square blocks out of $N \times N$ arrays, utilizing $m > N = 2^n$ memory modules, where *M* is a prime number. Their scheme allows the complicated full crossbar switch as the only possibility for data alignment circuitry and many costly *modulo(M)* operations with M not a power of two. In a publication, related to the development of the Burroughs Scientific Processor, Lawrie [8] proposes an alignment scheme with data switching, simpler than a crossbar switch, but still capable to handle only $\sqrt{N} \times \sqrt{N}$ square blocks out of *m=2N* modules, where $N = 2^{2n+1}$. Both schemes in [1] and [8] require larger number of modules than the number of simultaneously accessed (image) points (*N*). Furthermore, in both papers authors do not describe the necessary addressing circuitries for their schemes. Voorhis and Morin [12] suggest various addressing functions considering $p \times q$ subarray accesses and different number of memory modules M: both $m = p \times q$ and $m > p \times q$. Neither of the functions proposed in [12] is separable, which leads to an extensive number of address generation and module assignment logic blocks. In [3] authors propose a module assignment scheme based on Latin squares, which is capable of accessing $\sqrt{N} \times \sqrt{N}$ square blocks out of $N \times N$ arrays, but not from random positions. Similar drawbacks has the scheme proposed in [9]. One early graphical display system, described in [11], can be considered a partial case of our scheme, since authors describe square 8×8 submatrix accesses and memory alignment similar to the proposed in our scheme. The authors in [11] did not consider rectangular subarray accesses, which are not directly deducible from the proposed reading. No formalization of the addressing functions was presented either. A more recent display system memory, capable of simultaneous access of $p \times q$ rectangular subarrays is described in [10]. The design, proposed there, utilizes a prime number of memory modules, which enables accesses to numerous data patterns, but disallows separable addressing functions. Therefore, regarding block accesses, it is slower and requires more memory modules than our proposal. Large LUTs (in size and number) and yet longer critical path with consecutive additions can be considered as other drawbacks of [10]. A memory organization, capable of accessing

$N \times N$ square blocks, aligned into $(1+N)^2$ memory modules was described in [5]. The same scheme was used for the implementation of the matrix memory of the first version of HiPAR-DSP [14]. Besides the restriction to square accesses only, that memory system uses a redundant number of modules, due to additional DSP-specific access patterns considered. A definition of rectangular $p \times q$ block random addressing scheme from the architectural point of view dedicated for multimedia systems was introduced in [7], but no particular organization was presented there. In the latest version of HiPAR16 [4], the matrix memory was improved so that a restricted number of rectangular patterns could also be accessed. This design, however, still uses excessive number of memory modules as p and M respectively q and N should not have common divisors. E.g., to access the example 2×4 pattern, the HiPAR16 memory requires $3 \times 5 = 15$ memory modules, instead of 8 for our proposal. The memory of [4] would require more-complicated circuitry. Similarly to [11], [4], [14] assume separability, however, the number of utilized modules is even higher than the closest prime number to $p \times q$. Compared to [1], [3]–[5], [8]–[11], [14], our scheme enables higher scalability and lower number of memory modules. This reflects directly to the design complexity, which has been proven to be very low in our case. Address function separability reduces the number of address generation logic and critical path penalties, thus enables faster implementations. Regarding address separability, we differentiate from [1], [3], [8]–[10], [12], where address separability is not supported. As a result, *our design is envisioned to have the shortest critical path penalties among all referenced works.*

V. Conclusions

We presented a scalable memory organization capable of addressing randomly aligned rectangular data patterns out of a virtual 2D data storage. High performance is achieved by reduced number of data transfers between memory hierarchy levels, efficient bandwidth utilization, and short hardware critical paths. In the proposed design, data are located in an array of byte addressable memory modules by an addressing function, implicitly containing module assignment functions. An interface to a linearly addressable memory has been provided to load the array of modules. Theoretical analysis proving the consistency and efficiency of the linear and the two-dimensional addressing schemes has been also presented. The implementation of the organization was evaluated by experimental synthesis. Results indicate that a scalable range of such organizations can be efficiently mapped on recent FPGA technologies. At reasonably small hardware costs, we achieved considerable speedups of up to 8x for an experimental case study design versus traditional linearly addressable memories. The design is envisioned to be more cost-effective compared to related works reported in literature. The proposed organization is intended for specific data intensive algorithms in visual data processing applications, but can also be adopted by other general purpose applications with high data throughput requirements including vector processing.

ACKNOWLEDGEMENTS

This research is supported by PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, and the Technology Foundation STW (project AES.5021).

REFERENCES

[1] P. Budnik and D. J. Kuck. The organization and use of parallel memories. *IEEE Transactions on Computers*, 20(12):1566–1569, December 1971.

[2] ISO/IEC JTC11/SC29/WG11, N3312. MPEG-4 video verification model version 16.0.

[3] K. Kim and V. K. Prasanna. Latin squares for parallel array access. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):361–370, 1993.

[4] H. Kloos, J. Wittenburg, W. Hinrichs, H. Lieske, L. Friebe, C. Klar, and P. Pirsch. HiPAR-DSP 16, a scalable highly parallel DSP core for system on a chip: video and image processing applications. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 3112–3115, Orlando, Florida, USA, May 2002. IEEE.

[5] J. Kneip, K. Ronner, and P. Pirsch. A data path array with shared memory as core of a high performance DSP. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 271–282, San Francisco, CA, USA, August 1994.

[6] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.

[7] G. Kuzmanov, S. Vassiliadis, and J. van Eijndhoven. A 2D Addressing Mode for Multimedia Applications. In E. F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Workshop on System Architecture Modeling and Simulation (SAMOS 2001)*, volume 2268 of *Lecture Notes in Computer Science*, pages 291–306, SAMOS, Greece, July 2001. Springer-Verlag. ISBN: 3-540-43322-8.

[8] D. H. Lawrie. Access and alignment of data in an array processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, December 1975.

[9] D. lei Lee. Scrambled Storage for Parallel Memory Systems. In *Proc.IEEE International Symposium on Computer Architecture*, pages 232–239, Honolulu, HI, USA, May 1988.

[10] J. W. Park. An efficient buffer memory system for subarray access. *IEEE Transactions on Parallel and Distributed Systems*, 12(3):316–335, March 2001.

[11] R. F. Sproull, I. Sutherland, A. Thomson, S. Gupta, and C. Minter. The 8 by 8 display. *ACM Transactions on Graphics (TOG)*, 2(1):32–56, 1983.

[12] D. C. van Voorhis and T. H. Morrin. Memory systems for image processing. *IEEE Transactions on Computers*, C-27(2):113–125, February 1978.

[13] S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$-coded processor. In *11th International Conference on Field Programmable Logic and Applications (FPL)*, pages 275–285, Belfast, Northern Ireland, UK, August 2001.

[14] J. P. Wittenburg, M. Ohmacht, J. Kneip, W. Hinrichs, and P. Pirsh. HiPAR-DSP: a parallel VLIW RISC processor for real time image processing applications. In *3rd International Conference on Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97.*, pages 155–162, Melbourne, Vic. , Australia, December 1997.