A Graphics Parallel Memory Organization Exploiting Request Correlations

George Lentaris and Dionysios Reisis

Abstract—Real-time graphics applications require memory organizations featuring parallel pixel access and low-cost implementation. This work bases on a nonlinear skew mapping scheme and exploits the correlation between consecutive requests for pixels to design an efficient parallel memory organization. The mapping achieves parallel access, of mn pixels in various shapes, to the memory organized with mn banks. The proposed design technique combines the mapping properties and the spatial correlations among pixel requests to eliminate conflicts by spending at most one extra cycle every mn consecutive parallel pixel accesses. Consequently, the technique ensures that any pixel pattern—among these commonly used in graphics—can be accessed in a single cycle from any image location. The address computations become straightforward as the numbers of the requested pixels and the banks—apart from equal—can be powers of 2.

Index Terms—Parallel processing, graphics processors, storage devices, interleaved memories.

1 Introduction

International bibliography includes a plethora of strategies for organizing parallel memories. Each proposed parallel memory model has been designed to improve the speedup of the parallel algorithms, and consequently, the performance of the corresponding applications. The parallel memory design evaluation bases on criteria such as the number of included memory modules, the number of distinct data sets that can be accessed concurrently, the complexity of the addressing and the routing circuits involved, and the storage redundancy of each datum—most often zero. The design techniques which are considered of greater importance are those resulting in parameterized models suited to a relatively wide range of applications. Such techniques support a variety of different specifications and provide solutions to categories of problems such as the matrix/vector operations and the graphics processing.

The applications related to graphics most often require real-time performance. A solution to real-time requirements is parallel calculations accommodated by the parallel load and store of the corresponding pixels. The functionality and the level of parallelism of the chosen algorithm specify the subset of the image pixels requested by the process at each cycle. These access requests involve distinct pixel patterns (shapes) such as contiguous rows/columns, rectangular blocks, and even subsampled areas of the image. Subsampled areas are noncontiguous patterns, where the pixels are located at constant distances from each other. The majority of the graphics' algorithms generate memory requests for various patterns originating at any possible location on the image. To accommodate these needs, the memory designers have introduced solutions to the problems of minimizing the number of banks, designing effective bank/address calculation, and routing of the data. A well-known solution is to include a prime number of banks (e.g., [1], [6]) in the organization. However, this approach involves prime number calculations and leads to complicated addressing and bank selection circuits.

This work introduces a technique for designing parallel memory organizations with number of banks ranging from a power of 2 to any number for any application. The proposed technique avoids any prime number calculations and leads not only to efficient bank/address computations but also to straightforward designs for the routing network. The outcome of the design strategy fulfills the requirements of the graphics applications. Moreover, it includes and utilizes a minimum number of memory modules—equal to the number of requested pixels.

Compared to the hitherto published results, the proposed technique considers that the majority of the algorithms perform correlated image accesses. Related research on parallel memory organization for graphics applications relies on assumptions regarding only the shapes of the accessed pixel patterns. This approach is rational for an algorithm issuing requests assumed to be completely uncorrelated. However, uncorrelated requests are not usual in graphics applications. During the graphics processing, each algorithm follows specific strategies for requesting data. Each request usually addresses image locations depending on the data and/or the image locations accessed by the previous request. Even when the request sequence is not predefined but depends on calculations taking place during the process, a notable degree of correlation appears. Examples illustrating correlated requests can be found in common applications such as image filtering and video compression. A study of these examples shows that even the fast motion estimation algorithm, which is a highly unpredictable procedure, performs requests on a Macroblock basis.

A parallel memory organization can be significantly improved if we exploit the correlation of the requests performed in the course of a graphics application. This

• *The authors are with the Department of Physics, University of Athens, Panepistimiopolis Zografou, Physics Bld. IV, Athens 15784, Greece. E-mail: {glentaris, dreisis}@phys.uoa.gr.*

Manuscript received 31 July 2008; revised 20 Mar. 2009; accepted 17 July 2009; published online 11 Feb. 2010.
Recommended for acceptance by N. Ranganathan.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-07-0386.
Digital Object Identifier no. 10.1109/TC.2010.48.

work shows that the above statement is valid even for "generic" correlation assumptions. Such assumptions hold in many applications either because their algorithms behave accordingly or because the flow of the algorithms can be altered to follow the assumption without affecting their results. Specifically, we assume that the requests cover progressively a square region of the image/frame. Based on this, we give a technique for organizing the graphics' memory. The technique achieves conflict-free access, while it keeps the number of banks equal to the size of the access patterns and it avoids storing any redundant information. The proposed technique is advantageous because it removes the overhead of one extra memory module [3]. In the worst-case scenario, it introduces the probability of adding an extra clock cycle to those cycles required for covering the assumed square region. This extra cycle may or may not be required depending on the location of the square region within the image and on the shape of the access pattern. We consider this probability of an extra cycle as an improved overhead, especially if we consider the advantages gained by the evasion of prime numbers in the organization of the memory. The proposed organization supports the parallel access of all the widely used *contiguous* pixel patterns (rows, columns, rectangles), and moreover, the parallel access of a number of *sparse* patterns used in picture subsampling.

The remainder of the paper is organized as follows: Section 2 presents a review of the related published results. Section 3 introduces the proposed memory mapping and the notation to be used in the paper. Section 4 proves the accessibility of the proposed scheme. Section 5 describes how the proposed memory scheme exploits the request correlations in graphics algorithms. Section 6 shows example implementations of the memory organization and evaluates its performance in graphics applications. Section 7 analyzes the advantages of the proposed memory scheme, and finally, Section 8 concludes the paper.

2 Related Work

Related results to the organization of parallel memories have been published in the literature during the last four decades: Budnick and Kuck [1] introduced the *skewed schemes*, which perform simple arithmetic/modulo operations on the address of each array element to map the element to a memory module. Using modern notation [7], a *linear* skew scheme uses a mapping function of the form $module(i,j) = (a \cdot i + b \cdot j) \bmod (\mathcal{B})$, where (i,j) denotes the position of the element in the array and $\mathcal{B}$ denotes the total number of memory banks. The term *skew* was originally used to denote the displacement (shift) of two consecutive array rows in the memory, which results in from mapping data onto the banks. A linear skew scheme applies the same skew for each row/column of the array (i.e., *uniform skew*). Such schemes are *isotropic* [7], i.e., when two array elements are stored in the same memory bank, then each pair of their corresponding neighbors is also stored in the same bank, e.g., if (i_p, j_p) and (i_q, j_q) are stored in bank b_x, then (i_p+1, j_p+1) will be stored in the same bank with (i_q+1, j_q+1) (bank b_y). Besides linear skew, other similar schemes have been proposed in the literature. In some cases, the mapping scheme of a basic tile is repeated throughout the entire image [8]. In other cases, more complicated skews are applied, including *nonlinear* skews [2], [12], and *multiskewing* techniques [5], which use distinct linear mappings for different regions of the array. Tanskanen et al. [7] and Park [6] report and compare several different techniques for parallel memory organizations.

All the above skewing schemes—either linear or nonlinear—differentiate to each other with respect to the number of patterns that can be accessed in a single cycle (concurrently). These patterns define element sets, namely rows, columns, diagonals, triangles, squares, etc. In general, applications supporting large sets of different patterns require more complicated mapping functions. Moreover, when there is a requirement for fetching the patterns from any possible location on the array, they require a greater number of memory modules than the amount of a single pattern's elements (excess of banks): VanVoorhis and Morrin [2] prove that squares, columns, and rows cannot be accessed conflict-free when $\mathcal{B} = \mathcal{A}$, where $\mathcal{A}$ denotes the number of the pattern's data and $\mathcal{B}$ denotes the number of the memory's modules. A well-known solution to this problem is to fix $\mathcal{B}$ to a prime number greater than $\mathcal{A}$ (e.g., [1], [6]). Prime numbers though yield complicated address/mapping calculations, which result in great hardware resources overhead [2]. As an alternative solution, Liu et al. [11] propose a memory organization with $\mathcal{B} = 2 \cdot \mathcal{A}$ to avoid the use of prime numbers and improve the performance of [6]. However, the extended use of memory banks is not desirable because it leads to reduced bandwidth utilization and increases the area of the routing circuit. A consequent question is how to determine the minimum number of banks in a parallel memory system, when it is given the set of the required access patterns and their possible locations on the array. Chor et al. [3] address the above theoretic question for graphics applications, where the access patterns constitute rectangles of variable dimensions and are located at arbitrary positions on the image. They propose a doubly periodic assignment function placing the pixels on a *Fibonacci* lattice. Their organization guarantees that no conflicts occur when accessing areas of at most $\mathcal{B}/\sqrt{5}$ pixels. As an immediate result of [3] (also reported in other papers, e.g., [2]), only one extra memory module is required when the pattern set includes columns, rows, and squares at arbitrary locations on the image.

A different type of memory interleaving is achieved with XOR-based hash functions. *XOR-schemes* identify the memory bank storing a specific array element by computing the exclusive-or (XOR) of a subset of the element's address bits—or its (i,j) position in the array. Frailong et al. [9] introduced a general framework for describing a wide range of XOR-schemes; they also proposed a specific mapping, which allows conflict-free access to rows, columns, rectangles, and chessboards of a square array when the number of memory banks is an even power of 2. Since [9], many architectures have been proposed in the literature making use of various XOR-schemes in arrays' processing to achieve parallel access with specific properties [7]. Vandierendonck and De Bosschere [10] studied a simple methodology to design/choose the appropriate XOR hash function for an application. They propose two novel representations, different from the commonly used matrix representation of

TABLE 1
Classification of Parallel Memory Organizations

		Banks	Patterns	Proposed in	
Single Cycle Access	Unrestricted	$prime > \mathcal{A}$	r, c, d, d', b	[1], 1971	Linear
		$\mathcal{A}$+1, 2$\mathcal{A}$, $\mathcal{A}^{3/2}$	r, c, b	[2], 1978	
		$\mathcal{A}$+1, $\sqrt{5}\mathcal{A}$	r, c, b^*	[3] , 1986	
		$prime > \mathcal{A}$	r, c, d, d', b, s	[6], 2004	
		2$\mathcal{A}$	r, c, d, d', b, s	[11], 2007	Non-Linear
	Restricted	$\mathcal{A}$	r, c, b	[9], 1985	
		$\mathcal{A}$	r, c, b, s	[4], 1993	
		$\mathcal{A}$	r, c, b, b'	[7], 2005	

$\mathcal{A}$: number of pixels per access, r: rows, c: columns, d: diagonals, d': backward diagonals, b: blocks, b': inverted blocks, b^*: variable size blocks, s: sparse blocks

a hash function: the "null space" of the matrix, which simplifies the task of constructing a function that maps specific patterns without conflicts, and the "column space," which serves more elaborate tasks as, for example, the fan-in minimization of the XOR gates. Overall, XOR-schemes perform simple bitwise XOR-AND operations to map the data on the memory, and thus, they provide low latency (fast bank/address calculations) compared to the skew schemes [10]. However, the XOR technique results in nonisotropic mappings, which complicate the data alignment in applications using many access formats. Moreover, in applications requiring unrestricted access over periodic basic domains (e.g., graphics applications), the computation/interconnection circuits of the XOR-schemes deal with a greater amount of possible permutations compared to the isotropic mappings [7].

Besides the simple skew and the XOR schemes, other techniques have been presented in the literature for the organization of parallel memories, which aimed mainly in accommodating restricted array accessing. For instance, Kim and Prasanna [4] propose the use of Latin squares for the organization of a parallel memory; they introduced a methodology to construct *perfect* Latin squares, where no symbol appears more than once in any row, in any column, in any diagonal, or in any *main* subsquare of the array.

Finally, in the multiprocessors' cache-based hierarchies, besides the aforementioned techniques, two strategies are common: either the use of pseudorandom hash functions to reduce memory contention by uniformly distributing the bank addresses [13], [14] or the use of multiple hash functions which change during runtime to accommodate different access patterns [15]. The cache-based architectures have a slightly different goal compared to the architectures described in the above paragraphs: the noncache-based multiprocessors target the complete elimination of the conflicts so that predefined patterns can be accessed in a single cycle. The cache-based target the reduction of the cache misses, which increase the required access cycles.

A classification of the aforementioned parallel memory organizations is shown in Table 1 (the characterization unrestricted/restricted refers to the ability of accessing pixel patterns from any arbitrary location on the image, or not). The organization proposed in this paper should be placed at the bottom row of Table 1 (restricted, $\mathcal{A}$, $rcbs$, nonlinear). However, as it will become clear in Section 5, with the loss of only one cycle every $\mathcal{A}$ correlated requests, the proposed solution can be viewed as an organization permitting unrestricted access on the image by using only $\mathcal{A}$ banks.

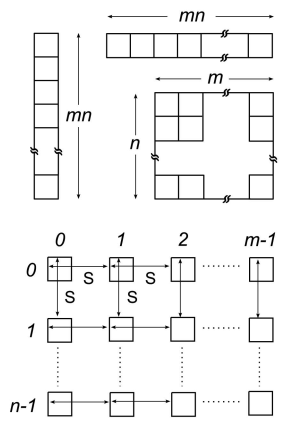
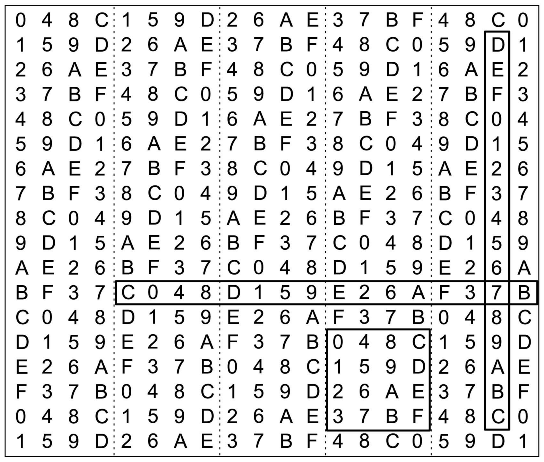


Fig. 1. Four *basic* access formats: *Column*, *Row*, *Block*, and *Sparse-s*.

3 Notation and Scheme Description

Throughout the paper, we refer to pixels on the image using coordinates (x, y), where x and y are integer numbers denoting the horizontal and vertical distances, respectively (alternatively, x represents the column and y the row). The parallel access of a pixel group assumes the use of a predefined access format. Let us define four *basic* formats: *Column*, *Row*, *Block*, and *Sparse-s*. Considering that all the above formats have a size of $\mathcal{A}$ pixels, we use m and n parameters to define the number $\mathcal{A}$ as $\mathcal{A} = m \cdot n$. Fig. 1 depicts the four formats. The first three formats, also referred to as *contiguous* patterns, are widely used in many research papers and applications. A *Sparse* pattern is characterized by the s parameter, which defines the actual space between the pattern's pixels. More specifically, s defines the distance in image pixels between two neighboring members of *Sparse* (when $s = 1$, we have a *contiguous* pattern). The *Sparse* format can be used to symmetrically subsample an area of the image. We generalize the notion of subsampling by using the term s_1-s_2 *Subsample* format, where s_1 defines the horizontal distance and s_2 defines the vertical distance between two neighboring members of this format. Each access format can be referenced from arbitrary positions on the image. We specify the position of a format on the image plane by referencing the position of its upper-left pixel on the image: *"formats originate at arbitrary (x, y)."* Finally, $\lfloor \frac{a}{b} \rfloor$ denotes the quotient of the integer division a *over* b, while $a \bmod (b)$ denotes the remainder; when the remainder of this division is zero, we use the notation $b \mid a$, i.e., b *divides* a.

In the proposed organization, the number of memory banks $\mathcal{B}$ is equal to $\mathcal{A}$, $\mathcal{A} = m \cdot n$. We use a function, which we call Φ, to map the pixels onto the memory banks. The input to this function is the location of a pixel on the image, (x, y), and the output is the identification number of a memory bank. The purpose of designing Φ is to divide the image in vertical stripes of width m and within each stripe to get a linear column skew by n. Furthermore, each pair of



0	4	8	C	1	5	9	D	2	6	A	E	3	7	B	F	4	8	C	0
1	5	9	D	2	6	A	E	3	7	B	F	4	8	C	0	5	9	D	1
2	6	A	E	3	7	B	F	4	8	C	0	5	9	D	1	6	A	E	2
3	7	B	F	4	8	C	0	5	9	D	1	6	A	E	2	7	B	F	3
4	8	C	0	5	9	D	1	6	A	E	2	7	B	F	3	8	C	0	4
5	9	D	1	6	A	E	2	7	B	F	3	8	C	0	4	9	D	1	5
6	A	E	2	7	B	F	3	8	C	0	4	9	D	1	5	A	E	2	6
7	B	F	3	8	C	0	4	9	D	1	5	A	E	2	6	B	F	3	7
8	C	0	4	9	D	1	5	A	E	2	6	B	F	3	7	C	0	4	8
9	D	1	5	A	E	2	6	B	F	3	7	C	0	4	8	D	1	5	9
A	E	2	6	B	F	3	7	C	0	4	8	D	1	5	9	E	2	6	A
B	F	3	7	C	0	4	8	D	1	5	9	E	2	6	A	F	3	7	B
C	0	4	8	D	1	5	9	E	2	6	A	F	3	7	B	0	4	8	C
D	1	5	9	E	2	6	A	F	3	7	B	0	4	8	C	1	5	9	D
E	2	6	A	F	3	7	B	0	4	8	C	1	5	9	D	2	6	A	E
F	3	7	B	0	4	8	C	1	5	9	D	2	6	A	E	3	7	B	F
0	4	8	C	1	5	9	D	2	6	A	E	3	7	B	F	4	8	C	0
1	5	9	D	2	6	A	E	3	7	B	F	4	8	C	0	5	9	D	1

Fig. 2. A 16-bank instantiation of the proposed scheme with $m = n = 4$.

consecutive stripes is linearly skewed by the least possible factor (i.e., one). The resulting mapping function Φ is the composition of the above two linear mappings—one for the columns and one for the stripes—and the idea of applying this is to maintain, at a reasonable level, the advantages of the linear skew schemes.

The proposed memory scheme is defined by the mapping function Φ:

$$\Phi(x,y) = \left(x \cdot n + y + \left\lfloor \frac{x}{m} \right\rfloor\right) \bmod (mn). \tag{1}$$

The above function uses the term $(y) \bmod (mn)$ to map the pixels of each column x onto the mn banks. The term $(x \cdot n) \bmod (mn)$ gives to the column x a skew of size n with respect to column $x-1$, if both columns $x-1$ and x belong to the same stripe. Finally, the term $(\lfloor \frac{x}{m} \rfloor) \bmod (mn)$ provides the skew between consecutive stripes. The use of the additive part $\lfloor \frac{x}{m} \rfloor$ characterizes Φ as a nonlinear scheme and yields some novel properties, which can be used to improve the processing performance of the graphics applications. The mapping Φ adapts to the number of available memory banks, $\mathcal{B}$. Fig. 2 shows an example mapping, where we consider a 16-bank memory with $m = n = 4$. Fig. 2 also depicts the cases of three specific patterns featuring conflict-free access (a *Block*, a *Row*, and a *Column*). We mention here that the authors of [1], [2] among the mappings they have presented, they have included Φ without though exploring its useful properties and the efficiency of the resulting organization.

4 Memory Accessibility

As reported in Section 2, Chor et al. [3] prove that the optimal solution for using any of the three *basic contiguous* formats lies in organizations using one extra bank. This section shows that the proposed organization, with exactly $\mathcal{B} = m \cdot n$ memory modules and $\mathcal{B}$ required bandwidth, minimizes the memory conflicts per memory access. More specifically, using Φ in the organization results in a maximum of one conflict per memory access. We prove that Φ, in the case of specific formats, permits conflict-free access unconditionally on the image. In other cases, Φ results in at most one conflict depending on the format's location on the image. These format types include both *contiguous* and *sparse* patterns. Moreover, we prove that when a conflict occurs during the memory access, it corresponds to a specific pixel of the accessed pattern: the down-right pixel of the pattern. This additional property of Φ simplifies the design of the circuit accommodating the memory organization.

In the following of this section, we study and show the accessibility of each *basic* format separately. The consequent proofs rely on the properties of a specific equation given in Lemmas 4.1 and 4.2. Lemma 4.1 focuses on the following problem: in which cases are two pixels (x_1, y_1) and (x_2, y_2) stored by Φ in the same bank? The lemma answers by solving the equation $\Phi(x_1, y_1) - \Phi(x_2, y_2) = 0$. The position of the pixel (x_2, y_2) is expressed with coordinates $(i \cdot s, j \cdot s)$ relative to the (x_1, y_1). The s parameter is used for the *Sparse-s* pattern (in the cases of *contiguous* patterns $s = 1$):

Lemma 4.1. *For any constant integers x, $n > 0$, $m > 0$, and any integer s with $s \mid m$, congruence (2)*

$$i \cdot s \cdot n + j \cdot s + \left\lfloor \frac{x + i \cdot s}{m} \right\rfloor - \left\lfloor \frac{x}{m} \right\rfloor \equiv 0 \pmod{mn} \tag{2}$$

has at most the following two solutions when $0 \le i < m$ and $|j| < n$:

$$(i,j) = (0,0)$$
$$\text{and} \quad (i,j) = (m-1, n-1).$$

Proof. See the Appendix. □

Lemma 4.2 is complementary to Lemma 4.1. It examines the case of the origin (x_1, y_1) located at specific coordinates (left border of any vertical stripe of the divided image) and shows that in this case, a second solution to (2) does not exist.

Lemma 4.2. *For any constant integers $m > 0$, $n > 0$, s with $s \mid m$, and x with $x \bmod (m) < s$, congruence (2) has exactly one solution when $0 \le i < m$ and $|j| < n$:*

$$(i,j) = (0,0).$$

Proof. See the Appendix. □

The following theorems make use of Lemma 4.1 to prove that the proposed memory scheme—using Φ—permits access to the four *basic* formats with at most one conflict. The corollaries of these theorems make use of Lemma 4.2 to determine the cases in which these formats can be accessed without conflicts. The first theorem considers the *Block* format:

Theorem 4.1 (Block). *Memory scheme Φ permits access to any $m \times n$ pixel block originating at the arbitrary (x, y) on the image with at most one conflict. If a conflict occurs, then it will appear at the down-right pixel of the block.*

Proof. Consider an $m \times n$ pixel block A_{xy} originating at the arbitrary (x, y) point on the image. For each pixel (x_p, y_p) within A_{xy}, we examine whether the remaining pixels of

A_{xy} reside in the same memory bank with (x_p, y_p). A pixel (x_q, y_q) resides in the same bank with (x_p, y_p) when

$$\Phi(x_p, y_p) = \Phi(x_q, y_q). \quad (3)$$

Expressing each (x_q, y_q) of A_{xy} with coordinates (i, j) relative to (x_p, y_p), from (3), we derive the congruence (2) of Lemma 4.1 with $s = 1$. Therefore, the solutions to (2)—besides $(i, j) = (0, 0)$—define the pixels of A_{xy} stored in the same bank with (x_p, y_p). According to Lemma 4.1, if (x_q, y_q) is a solution to (3), then this pixel will be located at distance $(i, j) = (m-1, n-1)$ from (x_p, y_p). Given the dimensions of A_{xy}, the only (x_p, y_p) for which the (x_q, y_q) solution of (3) corresponds to a pixel within A_{xy} is $(x_p, y_p) = (x, y)$ with $(x_q, y_q) = (x+m-1, y+n-1)$. Thus, memory scheme Φ assures that every pixel of A_{xy}, with the exception of the last pixel (at the down-right corner of A_{xy}), is stored in a distinct memory bank. Note here that during the examination of each pixel (x_p, y_p) within A_{xy}, it suffices to check (3) only against the pixels (x_q, y_q) of A_{xy} that are located to the right side, $i \geq 0$, of (x_p, y_p). When every (x_p, y_p) within A_{xy} is examined in this way, then every pixel within A_{xy} is checked against each other. □

In certain cases, the *Block* format can be accessed conflict-free. As the image has been divided into disjoint vertical stripes by Φ (Section 3), consider the case of the *Block* residing within a single vertical stripe (as shown, for example, in Fig. 2). Here, $x \bmod (m) = 0$ and Lemma 4.2 will apply with $s = 1$. In this case, even for the upper-left pixel of the arbitrary *Block*, there exists only one solution to (2). Hence:

Corollary 4.1. *Memory scheme Φ permits conflict-free access to any $m \times n$ pixel block originating at (x, y) on the image when $x \bmod (m) = 0$.*

Then, we focus on the access of the *Sparse-s* format.

Theorem 4.2 (Sparse-s). *When $s \mid m$, memory scheme Φ permits access to any Sparse-s pattern originating at the arbitrary (x, y) on the image with at most one conflict. If a conflict occurs, then it will appear at the down-right pixel of the Sparse-s pattern.*

Proof. See the Appendix. □

The accessing of the *Sparse-s* format can be performed conflict-free in certain cases. Whether or not there will be a conflict in such an access depends on the origin of the pattern and its sampling distance defined by the s parameter. If the origin of the pattern is relatively close to the left border of a vertical stripe (i.e., $x \bmod (m) < s$), then Lemma 4.2 applies and hence:

Corollary 4.2. *When $s \mid m$ and $x \bmod (m) < s$, memory scheme Φ permits conflict-free access to any Sparse-s pattern originating at (x, y) on the image.*

A format type which is accessed conflict-free anywhere (unconditionally) on the image plane is the *Sparse-m* format. In this case, $s = m$ and the condition $x \bmod (m) < s$ of Lemma 4.2 is always true. Lemma 4.2 applies for any arbitrary x and the derived congruence equation, for any pixel of the *Subsampled* block, has only one solution. Hence:

Corollary 4.3. *Memory scheme Φ permits conflict-free access to any Sparse-m pattern originating at the arbitrary (x, y) on the image.*

The following two theorems examine the cases of accessing the *Row* and *Column* formats. The *Column* format can be accessed conflict-free unconditionally on the image, due to the direct memory interleaving applied to each column of the image. Moreover, Φ skews the columns of the image to avoid the repetition of the memory banks in the horizontal direction and permit access to the *Row* format. Hence, the following hold:

Theorem 4.3 (Column). *Memory scheme Φ permits conflict-free access to any column of length mn located at the arbitrary (x, y) on the image.*

Proof. See the Appendix. □

Theorem 4.4 (Row). *Memory scheme Φ permits conflict-free access to any row of length $mn - 1$ originating at the arbitrary (x, y) on the image.*

Proof. See the Appendix. □

Furthermore, any *Row* pattern of length mn can be accessed conflict-free when it originates at the left border of a vertical stripe of the image (as shown, for example, in Fig. 2). This is true because when $x \bmod (m) = 0$ Lemma 4.2 applies, and thus, even for the origin pixel of the *Row*, congruence (2) has only one solution pointing within the row. Hence:

Corollary 4.4. *Memory scheme Φ permits conflict-free access to any row of length mn originating at (x, y) on the image when $x \bmod (m) = 0$.*

Similar arguments to those used to prove the above theorems can be used to show the accessibility of the proposed memory scheme to even more elaborate—and not so widely used—access patterns. This section concludes by showing the accessibility of a specific access pattern, which we call *multisquare-r*. The importance of studying this pattern will be shown in the following section, where we will describe a technique to handle conflicts efficiently.

We define as *multisquare-r* a pattern consisting of regularly scattered squares on the image (we consider this as an auxiliary format to distinguish from the *basic*). In such a pattern (Fig. 3), the size of each square is $r \times r$ pixels and the squares are placed on a lattice with basis vectors $\widehat{a} = (r \cdot m, 0)$ and $\widehat{b} = (0, r \cdot n)$, where $r \mid m$ and $r \mid n$. A *multisquare* consists of $\frac{n}{r} \times \frac{m}{r}$ squares so that the total number of its pixels is $n \times m$. Note that this pattern consists of $n \times m$ pixels instead of $m \times n$, which is the case of the *basic* patterns.

Theorem 4.5 (Multisquare). *A multisquare-r originating at the arbitrary (x, y) on the image can be accessed conflict-free when $x \bmod (m) \leq m - r$.*

Proof. See the Appendix. □

We can highlight the correctness of this theorem by describing the construction of the pattern. Starting from a conflict-free accessible *Block* (Corollary 4.1), we divide this

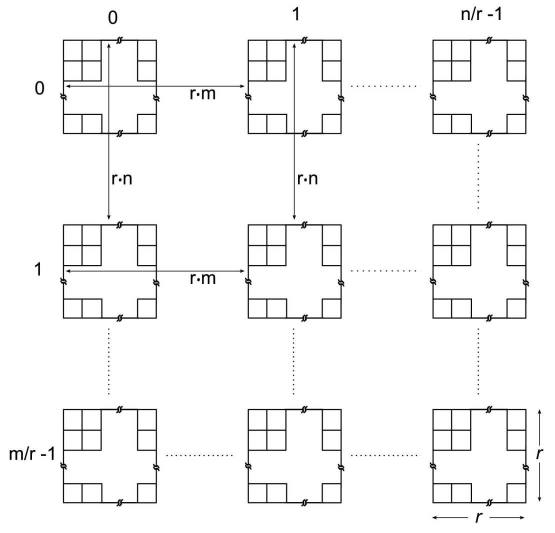


Fig. 3. Auxiliary access format: *multisquare-r*.

Block into subblocks of size $r \times r$ each. The idea is to exchange each subblock, R, of the *Block* pattern with another $r \times r$ block, R', at a distant location of the Φ plane such that both subblocks R and R' have their corresponding pixels mapped to the same banks.

An immediate result of Theorem 4.5 is that the *m-n Subsample* of any $m \cdot n \times m \cdot n$ pixel region can be conflict-free accessed unconditionally on the image. Note that the *m-n Subsample* pattern is a *multisquare* with $r = 1$ (*multi-square-1*). For any *multisquare-1* pattern, the hypothesis of Theorem 4.5 is always true (i.e., $x \bmod (m) \leq m - 1$) and:

Corollary 4.5. *Memory scheme Φ permits conflict-free access to the m-n Subsample of any $m \cdot n \times m \cdot n$ pixel region originating at the arbitrary (x, y) on the image.*

5 Multiple-Format Conflict-Free Access to *MacroSquares*

The previous section has presented the efficiency of the proposed memory scheme Φ, and showed that Φ can access mn pixels at arbitrary locations on the image with either none or at most one conflict depending on the access format and its location. This section presents a technique exploiting Φ's properties to avoid the above conflicts and provide access of the mn pixels in a single cycle. The presented technique takes into account the correlation between consecutive pixel requests in graphics applications. The assumption is that in the course of processing an image/frame, the algorithm does not access data from positions which are unrelated to each other, but the sequence of requests covers progressively a square region of arbitrary origin. The term generic seems appropriate for this assumption since it holds for many applications and many applications can adapt to it.

In the following paragraphs, we will use the above assumption to work on square regions of $W \times W$ pixels. We show that an efficient solution for accessing such square regions can be obtained for $W = mn$. Such an $mn \times mn$ pixel region will be denoted by a *MacroSquare* originating at the arbitrary (x, y) on the image. Further, the term *scan* of a region will be referring to the sequence of accesses covering the entire pixel region.

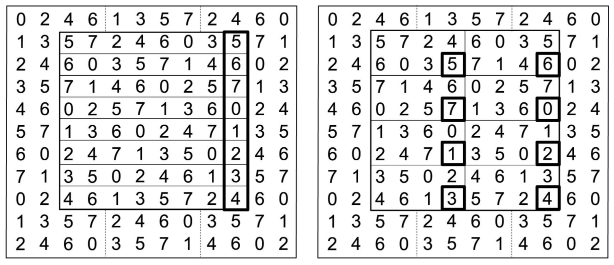


Fig. 4. Scanning an 8×8 *MacroSquare* with the *Row* and the *Block* formats.

The proposed technique optimizes the cycle count of the *MacroSquare* scan process when conflicts occur during the pattern requests. In such cases, a straightforward application of the scan process would take more than $mn + 1$ cycles. Examples of such cases are depicted in Fig. 4, where we use two different *contiguous* formats to scan a *MacroSquare* in an 8-bank organization with $m = 4$ and $n = 2$. We observe that the conflict pixels (marked here with heavy lines) are stored in distinct memory banks, and thus, we can access them concurrently, i.e., in a single cycle. The same holds for the conflict pixels when we use the *Sparse-2* format to scan this specific *MacroSquare* (Fig. 5). We refer to the conflict occurring during an access cycle of a region scan as a *local conflict* of that cycle. To each *local conflict* correspond an unaccessed pixel of the requested pattern and an unused memory bank (idle) called hereafter the *missing pixel* and the *missing bank* of the requested pattern.

We will prove that Φ permits conflict-free access to any arbitrary *MacroSquare* on the image in at most $mn + 1$ cycles, using **any** of the four *basic* access formats of Φ. Moreover, we will show that **no** extra buffers are required for this operation. This statement can be considered as dual to the statement derived from the optimal solution presented in [3]: "any arbitrary $n^2 \times n^2$ *MacroSquare* on the image can be accessed conflict-free in at most n^2 cycles, using $n^2 + 1$ memory banks" (consider $m = n$ for this comparison).

The following lemma determines the *missing bank* of the pattern when a *local conflict* occurs (consider $s = 1$ for the *contiguous* patterns):

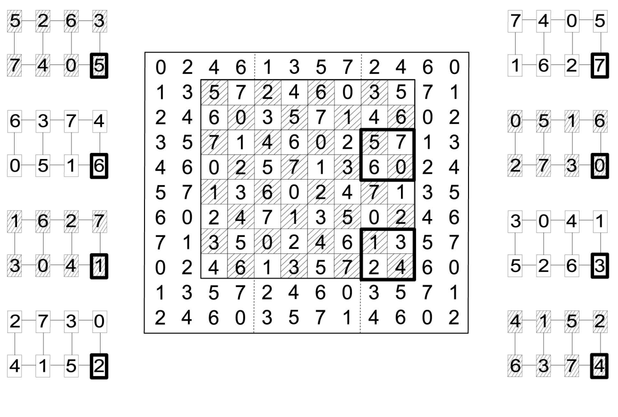


Fig. 5. Scanning an 8×8 *MacroSquare* with the *Sparse-2* format.

Lemma 5.1. *The missing bank of a basic access format originating at the arbitrary (x, y) on the image is given by the mapping*

$$mbank\,(x, y, s) = \Phi\Big(\Big\lfloor\frac{x}{m}\Big\rfloor m + (x) \bmod (s), y\Big). \qquad (4)$$

Proof. See the Appendix. □

The above proof uses the idea that in the case of a pattern request causing a *local conflict*, there is no pixel of the pattern mapped by Φ to the *missing bank*. In other words, it shows that $mbank(x, y, s) \neq \Phi(x_q, y_q)$ for any pixel (x_q, y_q) of the pattern.

Then, we prove that the *missing pixels* during a scan are located in distinct banks. We begin by defining as *typical* the scan of a region containing a multiple of mn pixels when: 1) the *basic* access format does not change during the scan, 2) each and every pixel of the region is requested exactly once, and 3) none of the requested pixels lies outside the region. Note that any *contiguous* format can be used for a *typical MacroSquare* scan process, while the use of a *Sparse-s* format is feasible only when $s \mid m$ and $s \mid n$. *Typical MacroSquare* scan examples are depicted in Figs. 4 and 5.

Theorem 5.1. *For any typical MacroSquare scan which includes local conflicts, the following hold:*

1. *Each missing pixel is stored in a distinct memory bank.*
2. *Each missing bank is unique.*

Proof. The *Column* and the *Sparse-m* formats never lead to *local conflicts* (Theorem 4.3 and Corollary 4.3). We therefore study a *typical MacroSquare* scan with the use of the remaining *basic* formats: *Row*, *Block*, and *Sparse-s* with $s \neq m$. We begin with part 1 of the theorem as follows:

1. *Missing pixels*

   Row: using Theorem 4.4, we deduce that the *missing pixels* of a *typical MacroSquare* scan—using the *Row* format—are located at the rightmost column of the *MacroSquare*. These pixels constitute a *Column* pattern, and therefore, according to Theorem 4.3, they are always stored in distinct banks.

   Block: using Theorem 4.1, we deduce that the *missing pixels* of a *typical MacroSquare* scan—using the *Block* format—are located at distances (m, n) from each other. These pixels constitute an *m-n Subsample* of the *MacroSquare*, and therefore, according to Corollary 4.5, they are always stored in distinct banks.

   Sparse-s: in this case, the number of the *MacroSquare*'s *missing pixels* varies from m to mn, depending on the *MacroSquare*'s origin on the image. The down-right pixels of the mn *Sparse-s* patterns constitute a *multisquare-s*. Using Theorem 4.2, we deduce that the pixels of this *multisquare* are the only candidate *missing pixels* of the *MacroSquare*. To determine which of these pixels are actually *missing pixels*, we use Corollary 4.2. More specifically, when a *Sparse-s* request originates at (x_{req}, y_{req}) on the image, then the pattern's down-right pixel is located at $(x_{pix}, y_{pix}) = (x_{req} + s \cdot (m-1), y_{req} + s \cdot (n-1))$. Using Corollary 4.2, we deduce that the down-right pixel of the *Sparse-s* pattern is a *missing pixel* when $x_{pix} \bmod (m) < m - s$. To conclude the *Sparse-s* case, we examine two different subcases as imposed by the hypothesis of Theorem 4.5. In the first subcase, the *multisquare* originates at (x_{sqr}, y_{sqr}) with $x_{sqr} \bmod (m) \leq m - s$. Theorem 4.5 guarantees that all of the *multisquare*'s pixels are stored in distinct memory banks (no need to determine the conflicts in this subcase). In the second subcase, we divide the *multisquare*'s pixels into two sets: $\mathcal{P}_m$ and $\mathcal{P}_h$. The $\mathcal{P}_m$ set includes the pixels with the property $x_{pix} \bmod (m) < m - s$, i.e., the *missing pixels* of the *MacroSquare*. The $\mathcal{P}_h$ set includes the remaining pixels of the *multisquare*, i.e., the non*missing pixels*. We are interested only in the $\mathcal{P}_m$ set. $\mathcal{P}_m$ misses certain pixel columns compared to the *multisquare* examined. More specifically, each of the squares constituting the examined *multisquare* misses its left vertical stripe of width $(-x_{sqr}) \bmod (m)$, the pixels of which have the property $x_{pix} \bmod (m) \geq m - s$. Therefore, we can consider these rectangles to be part of another distinct *multisquare* originating at $(x'_{sqr}, y'_{sqr}) = (\lfloor\frac{x_{sqr}}{m}\rfloor m + m, y_{sqr})$. Theorem 4.5 guarantees that no two pixels of the (x'_{sqr}, y'_{sqr}) *multisquare* are stored in the same bank.

2. *Missing banks*

   According to Lemma 5.1, if a request originating at (x, y) on the image leads to a *local conflict*, then its *missing bank* will correspond to the bank storing the pixel located at $(\lfloor\frac{x}{m}\rfloor m + (x) \bmod (s), y)$ on the image. We will denote this pixel as the *representative* pixel of the request's *missing bank*. We show that the *typical MacroSquare* scan results in a set of *missing banks* whose representative pixels are stored in distinct memory banks (this is equivalent to proving the uniqueness of the *missing banks*).

   Row: for a *typical* scan using the *Row* format, we express the origins of the mn requests with coordinates (i, j) relative to (x, y), where (x, y) is the origin of the *MacroSquare*. The requests' origins can be referenced by $(x, y + j)$, with $j \in [0, mn - 1]$. According to Lemma 5.1, the *missing banks* of these requests store the pixels located at $(\lfloor\frac{x}{m}\rfloor m, y + j)$, with $j \in [0, mn - 1]$. These pixels form a *Column* pattern originating at $(\lfloor\frac{x}{m}\rfloor m, y)$, and according to Theorem 4.3, they are stored in distinct memory banks.

   Block: using the same argument as above, the origins of the mn *Block* requests can be referenced by $(x + i \cdot m, y + j \cdot n)$, with $i \in [0, n-1]$ and $j \in [0, m-1]$. According to Lemma 5.1, the *missing banks* of these requests store the pixels located at $(\lfloor\frac{x}{m}\rfloor m + i \cdot m, y + j \cdot n)$, with $i \in [0, n-1]$ and

$j \in [0, m-1]$. These pixels form a *multisquare-1* pattern originating at $(\lfloor \frac{x}{m} \rfloor m, y)$, and according to Corollary 4.5, they are stored in distinct memory banks.

Sparse-s: the origins of the mn *Sparse-s* requests can be referenced by $(x+i', y+j')$, where $i' = \lfloor \frac{i}{s} \rfloor sm + (i) \bmod (s)$ and $j' = \lfloor \frac{j}{s} \rfloor sn + (j) \bmod (s)$, with $i \in [0, n-1]$ and $j \in [0, m-1]$. As mentioned before, in this case, the number of *local conflicts* varies with respect to the *MacroSquare*'s origin. According to Corollary 4.2, we are interested only in those *Sparse-s* requests for which $(x + \lfloor \frac{i}{s} \rfloor sm + \ (i) \bmod (s)) \bmod m \geq s$, and hence, the requests for which

$$(x \bmod m + i \bmod s) \bmod m \geq s.$$

Therefore, we examine only those values of i for which we have $s \leq x \bmod m + i \bmod s < m$. Using this inequality in the mapping (4), we deduce that the *missing banks* of the *Sparse-s* requests store the pixels located at $(\lfloor \frac{x}{m} \rfloor \ m + \lfloor \frac{i}{s} \rfloor \ sm + (x+i) \bmod s, y + \ \lfloor \frac{j}{s} \rfloor sn + j \bmod s)$, where $j \in [0, m-1]$ and $0 \leq i \leq n-1$. Note that: 1) i might not cover its entire domain because of the aforementioned constraint and 2) the above formula describes a (or part of a) *multisquare-s* originating at $(\lfloor \frac{x}{m} \rfloor m, y)$. Theorem 4.5 guarantees that no two pixels of this *multisquare* are stored in the same memory bank.□

As an immediate result of Theorem 5.1, the proposed memory scheme permits access to an arbitrary *MacroSquare* in at most $mn+1$ cycles. More specifically, when the selected format can be accessed conflict-free unconditionally on the image (e.g., *Column* or *Sparse-m*), then the *MacroSquare* scan requires exactly mn cycles. The same holds for any selected format when the *MacroSquare* originates at (x, y) on the image with $x \bmod m = 0$, because no *local conflicts* occur. When none of the above is the case, we can scan (read) the *MacroSquare* in exactly $mn+1$ cycles as follows: first, we retrieve all of the pixels that correspond to the forthcoming *local conflicts*, and afterward, during the mn accesses, we use them to replace the *missing pixels* of the requests. Theorem 5.1 guarantees that we can retrieve all of the forthcoming *missing pixels* in a single cycle. Moreover, Theorem 5.1 guarantees that no temporary buffers are required to store these *missing pixels*. Since the *missing banks* of a *MacroSquare* are unique, then in a single cycle, we can store each *missing pixel* to the corresponding *missing bank* of each future request (e.g., at the last memory address) and hence, for the remaining mn accesses, we avoid all *local conflicts*. Similarly, when the *MacroSquare* scan refers to write operations, we use the first cycle to write the forthcoming conflicts, and during the following mn cycles, we complete the write access.

The statements of Theorem 5.1 hold even when the access format changes during the *MacroSquare* scan. Such a format change can be envisaged as a composition of two (or more) distinct, nonoverlapping, *typical* region scans. For each of these scans separately, the statements of Theorem 5.1 hold because they constitute parts of *typical MacroSquare* scans. To show that the composition of the two nonoverlapping scans does not result in duplicate *missing banks* or duplicate banks storing the *missing pixels*, we use the following idea: assume that we perform two *typical* scans of an arbitrary pixel region (less than or equal to a *MacroSquare*) using two distinct *basic* access formats, namely $\mathcal{F}_1$ and $\mathcal{F}_2$. Employing the ideas used in the above proofs, we can show that the $\mathcal{F}_1$ *missing pixels* are stored in the same set of memory banks storing the $\mathcal{F}_2$ *missing pixels*. More specifically, we reference these *missing pixels* with coordinates relative to the origin of the scanned region and we apply mapping Φ to them. This process results in two bank sets, $\mathcal{S}_1$ and $\mathcal{S}_2$, which define the banks storing the $\mathcal{F}_1$ and $\mathcal{F}_2$ *missing pixels*, respectively. At this point, it is straightforward to show with isomorphisms that $\mathcal{S}_1 = \mathcal{S}_2$ (or, in the case of the format $\mathcal{F}_1$ leading to less conflicts than $\mathcal{F}_2$, $\mathcal{S}_1 \subset \mathcal{S}_2$). Furthermore, by applying mapping (4) instead of mapping Φ, we can show that the *missing banks* are also the same for the two region scans. Consequently, the *typical* scan of a region results in the same conflict banks for any selected access format; this fact allows the change of the format during the *MacroSquare* scan without violating the statements of Theorem 5.1.

To conclude this section, we mention that a *MacroSquare* can be accessed in exactly mn cycles using an access format different from the four *basic*. Such an example is the *multisquare-1* format, which, according to Corollary 4.5, leads unconditionally to no *local conflicts*. Furthermore, we note that the square region request correlation is not the only case where Φ maintains its *missing pixels* property. Consider, for example, the *typical* scan of an $m^2 n \times n$ horizontal stripe with the *Block* and the *Row* formats: it is straightforward to show, first, that the *missing pixels* of the *Block* requests are located at horizontal distances m from each other. Thus, the additive factor $\lfloor \frac{x}{m} \rfloor$ of Φ guarantees that these pixels are stored in distinct memory banks. Second, the *missing pixels* of the *Row* requests form columns of depth n located at distances mn from each other. Given the distance between them, the additive factor $\lfloor \frac{x}{m} \rfloor$ of Φ guarantees that these columns are linearly shifted to each other exactly by n, and thus, they involve distinct memory banks. In other examples, the *missing pixels*' property holds for Φ even when the scan is *nontypical*: consider mn successive vertical *Block* requests which are offset by only one pixel (e.g., a scan used in 2D image filtering). In this case, the *missing pixels* form an mn *Column* pattern, and thus, they are stored in distinct memory banks.

6 Applications with Φ and Performance Evaluation

This section shows example applications of the Φ memory organization. Further, it presents simulation results with respect to the number of access cycles required when Φ is used in real-world applications. In the following of the section, we include examples showing certain cases in which Φ features the same performance with the hitherto published solutions utilizing an extra [3], [2], a prime [1], [6], or a double number of banks [11]. We also present applications involving memory conflicts in which we measure the cycle overhead imposed by the correction technique of Section 5. The demonstration examples fall into two major categories of graphics applications, namely video compression and image filtering.

6.1 Application in Real-Time Motion Estimation

In a video encoder, the subsystem performing the most intensive computations is the Motion Estimation (ME) processor. During the ME process, each macroblock (a fixed square of 16×16 pixels) of the current frame is subtracted from various regions of the reference frame. The sequence of these operations is controlled by a "fast motion estimation algorithm," which matches the frame blocks according to their similarities. A common practice in real-time implementations of the ME process is the use of a memory buffer storing the macroblock and its corresponding search area [16]. The use of this buffer accelerates the searching procedure by allowing parallel access to the pixels of interest. The proposed mapping Φ can be efficiently utilized to serve the above purpose. For proof of concept, let us examine the case of an H.264 application with specified memory bandwidth of 8 pixels per cycle. Also, assume that this example application makes use of almost every H.264 feature: multiple reference frames, macroblock partitions, and half-pixel interpolations. We will demonstrate how Φ supports such functionality without delaying the algorithm, i.e., performing as the time-efficient solution [3] to the memory organization problem.

We organize the 8-bank buffer using Φ with $m = 4$ and $n = 2$ (Fig. 4). This scheme permits conflict-free access (Section 4) unconditionally on the image to the *Sparse-4* and the *4-2 Subsample* (i.e., *multisquare-1*) formats. We can scan any 16×16 pixel region of the search area in 32 clock cycles by using the *Sparse-4* format: we sequentially access 8-tuples of distinct pixels until the entire 16×16 region is covered. Similarly, using the *4-2 Subsample* format, we can scan any 8×8 region in 8 clock cycles. Therefore, regardless of the algorithm's choices on candidate targets and partitioning techniques, Φ can serve the pixel requests efficiently (100 percent bandwidth utilization). It is worth mentioning here that scanning a region with a sparse format has a twofold advantage over the contiguous format scans. First, it allows the algorithm to use only a subsampled part of the frame during the search process (e.g., chessboard). Second, the use of scattered samples instead of solid blocks leads to more accurate estimations of the region's final metric value during the subtraction procedure. These estimations are important to techniques such as the "early termination" and the "speculative execution" [16], which speed up the algorithm in real-time applications. To complete the example, we examine the stage of the ME process, where the algorithm requests half-pixels from the buffer. The H.264 standard defines a 6-tap FIR filter for pixel interpolations [17]. To support this function, we use the *Row* and *Column* formats: the *Row* format for generating horizontal displacements and the *Column* format for generating vertical displacements. It is noteworthy that Φ imposes the same number of access cycles with the time-efficient 9-bank organization (a linear skew by 5) [3]. More specifically, to generate an entire interpolated row/column of a 16×16 region, the FIR filters will use $2 + 16 + 3 = 21$ distinct pixels. These pixels can be accessed in exactly three cycles by using the 9-bank organization. The same holds for Φ because it permits parallel access to at least seven pixels of the *Row*/*Column* format from anywhere on the frame (Section 4). Similarly, for an 8×8 region interpolation, the FIR filters operate on $2 + 8 + 3 = 13$ distinct pixels per row/column of the region. In this case, both Φ and the 9-bank organization lead to two access cycles per row/column of the interpolated region. To conclude, in this example application, we can make use of Φ's properties to construct a more cost-effective memory organization than those proposed in the literature, without cutting back on the performance of the algorithm. In the following sections, we describe two example applications leading to conflicts and we measure the cycle overhead imposed by the conflict handling technique described in Section 5.

6.2 Application in High-Definition Real-Time Motion Estimation

In this example application, we extend the use of the above ME module to high-definition video encoders. Here, the required memory bandwidth is 16 pixels per cycle (double than the previous example) and we organize the 16-bank buffer using Φ with $m = n = 4$ (Fig. 2). For simulation purpose, the stages of the block matching algorithm have been determined as follows: first, the algorithm performs a "Three-Step Search" to find an integer pixel matching for the entire 16×16 macroblock. Second, it refines the result of the first stage by partitioning the macroblock and searching for further displacements of its submacroblocks. Specifically, for each 8×8 partition, it performs a "Diamond Search" starting from the motion vector of the first stage. Third, it refines the results of the second stage by using half-pixel interpolations: for each submacroblock, it performs a local search in the vicinity of the motion vector of the second stage (when no submacroblock displacements are found in the second stage, it performs a half-pixel search for the entire macroblock).

When $m = n = 4$, Φ permits conflict-free access to the *Sparse-4* format unconditionally on the frame (Section 4). Therefore, during the first stage of the algorithm, we use the *Sparse-4* format to scan any requested 16×16 region in 16 cycles without delaying the process. During the second stage of the algorithm, we use the *Sparse-2* format to scan any requested 8×8 pixel region in four cycles. However, when these requests originate at positions (x, y) with $x \bmod 4 \neq 0$, we must use one extra cycle to retrieve the forthcoming conflicts of the scan process (as described in Section 5). During the third stage of the algorithm, we use the *Column* and *Row* formats to feed the 6-tap FIR filters generating the requested half-pixels. For the same reason described in the previous ME example, the use of Φ in the third stage of the algorithm does not result in more access cycles than the use of a time-efficient 17-bank organization [3]: for the generation of 16 half-pixels (an interpolated row/column), two access cycles are required, while for the generation of 8 half-pixels, one access cycle is required.

We developed a bit accurate model of the above ME processor and measured the number of memory accesses in 12 test sequences. The test sequences involved four well-known videos (namely Pedestrian, Rush Hour, Blue Sky, and River Bed, with 100 frames each) in three distinct frame resolutions: 720×576, $1{,}280 \times 720$, and $1{,}920 \times 1{,}088$. To determine the access overhead imposed by the conflict handling technique, we also measured the number of memory accesses assuming a 17-bank organization [3] for the processor's memory. More specifically, we calculate the cycle overhead of Φ as the percentage $\frac{C_\Phi - C_O}{C_\Phi}$, where C_Φ and C_O denote the access cycles imposed by the proposed 16-bank and the time-efficient 17-bank organizations, respectively. Table 2 presents the simulation results separately for each

TABLE 2
Performance of the Proposed Memory organization ($m = n = 4$) in Motion Estimation Applications

	MB Partitioning Procedure		**Overall Performance**	
Frame Size	*Cycles per Scan*	*Overhead (%)*	*Frame Accesses*	*Overhead (%)*
720×576	3.53	16.2	1.1×10^6	3.83
1280×720	3.68	15.9	2.7×10^6	3.59
1920×1088	3.77	15.6	6.4×10^6	3.49

frame resolution (table rows). The fourth and fifth columns of Table 2 present the statistics of the memory accesses with respect to the entire execution period of the algorithm (accesses per frame and accessing overhead of Φ). The third column presents the accessing overhead of Φ during the second stage of the algorithm (i.e., during the macroblock partitioning), where the proposed organization requires more access cycles than the extra bank organization [3] (as described above, we consider no overhead during the first and third stages of the algorithm). The second column shows the average number of cycles required for a *typical* 8×8 region scan of the search area, taking into account the commonly used "early termination" technique (stop scanning when the accumulation of the metric value exceeds a lower bound). The "early termination" technique leads to less than 4 cycles per scan, and therefore, it increases the expected cycle overhead of the conflict handling technique: the correction cycle accounts for less than every $4 + 1$ cycles. However, note that this increase is counterbalanced by the following fact. The percentage of the requested 8×8 regions, which require conflict handling, is less than the expected $3/4$ due to the zero biased nature of the block matching algorithms (approximately, this percentage is equal to 69 percent). Table 2 shows that the overhead decreases as the video resolution increases primarily due to the increase of the scan cycles, which indicates a higher correlation between successive pixel requests. This is because the "early termination" technique becomes less efficient in high-definition videos, where the pixels feature greater spatial correlations with each other. The overall evaluation of the memory performance showed that Φ features a notable degree of overhead improvement when compared to the bank overheads of the hitherto published solutions. Table 2 shows that Φ features an overall cycle overhead of less than 3.83 percent, while [1], [2], [3], and [6] feature a bank overhead of $1/17$, i.e., 5.9 percent ([11] features an even greater bank overhead of 50 percent).

6.3 Application in Image Filtering

In the last example, we consider a well-known field of graphics applications, namely the Image Filtering. Images are filtered in the spatial or in the frequency domain [21], depending on the purposes and the processing power of each application. When filtering in the spatial domain, a 2D array of coefficients ($K \times K$ filter kernel, for $K = 3, 5, \dots$) slides across the image, determining the operations to be performed on the pixels of each region. When filtering in the frequency domain, the image is forward transformed, multiplied by the frequency response of the filter, and finally, inverse transformed to generate the filtered output. The 2D transformation of a $W \times H$ image is most often performed by $H + W$ consecutive 1D transformations in the following order: first, H row transformations are performed on the image pixels, and afterward, W column transformations are performed on the transformed rows. Let us assume an application implementing various filtering techniques in both domains—spatial and frequency. A parallelization for this application would require memory access to three distinct formats: 1) *Blocks* from anywhere on the image, 2) nonoverlapping *Rows*, and 3) nonoverlapping *Columns*. The *Blocks* will be used during the sliding of the spatial kernel to fetch the pixels participating in the local filtering process. The *Rows/Columns* will be used to collect the entire row/column of the image, which, in turn, will be forwarded to the 1D transformation module. We organize the memory of this application using Φ with $m = n = K$, i.e., using a number of banks equal to the size of the application's filter kernel. During the frequency filtering, no conflicts occur because the nonoverlapping *Row/Column* requests are located at distances mn from each other (with $x \bmod m = 0$). During the spatial filtering, we will slide the *Block* format in antiraster scan order (note that for a raster scan, we can use a transposed version of Φ, obtained by exchanging x with y in the expression of the Φ function). Therefore, the down-right pixels (potential missing pixels) of mn consecutive *Block* requests will constitute a *Column* pattern. Based on this fact, we will use one cycle every mn requests to access the forthcoming missing pixels of the mn *Blocks*—when $x \bmod m \neq 0$.

Next, we describe the simulation results of the above operations. We use Φ to organize the memory and we measure the total number of access cycles, C_Φ, required to complete the filtering procedures. Also, we measure the cycles, C_O, required in the case of using the time-efficient solution [3]. Based on the results, we calculate the cycle overhead of Φ as the percentage $\frac{C_\Phi - C_O}{C_\Phi}$. Fig. 6 depicts the cycle overhead of Φ versus the number of required pixels per cycle (the size of the filter's kernel in different applications). This figure depicts the following curves: the solid line (ii) represents the mean value of the cycle overhead over the spatial and frequency domain procedures. Curve (iii) represents the cycle overhead when filtering in the spatial domain. Curve (iv) represents the cycle overhead when filtering in the frequency domain. We have to mention here that even though there are no conflicts when filtering in the frequency domain with the Φ organization, we consider a cycle overhead due to the ability of [3] to access $mn + 1$ pixels per *Row/Column*. The simulations showed that the variation of the Φ overhead is limited (it approaches 0) with respect to the size of the image, and thus, we do not include such

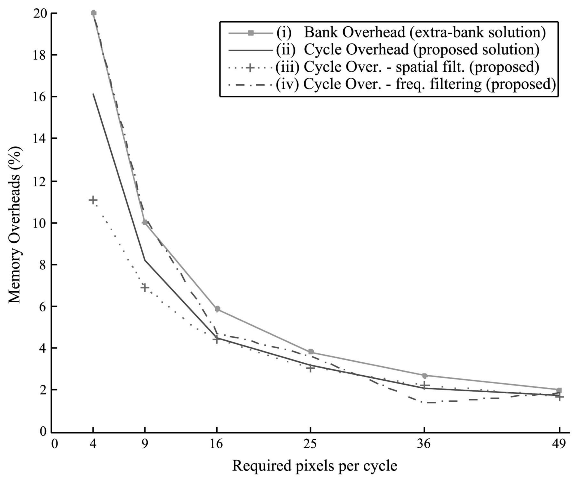


Fig. 6. Comparing the cycle overhead of Φ to the bank overhead of [3] in filtering applications.

information here. Fig. 6 also depicts curve (i), which represents the bank overhead of [3] (one extra bank) as an indication of the hardware resource savings of Φ. Note that the Φ overhead is always less than the extra bank overhead of the time-efficient solution presented in [3] (and also less than the prime bank overheads of [1], [6], which, in turn, are greater than or equal to [3]).

7 Advantages of the Proposed Organization

Φ fulfills the requirements of graphics applications and at the same time keeps the number of banks equal to the number of pixels accessed at each cycle. Compared to the hitherto published techniques, which involve a number of banks either prime or double than the accessed pixels, Φ improves the complexity of the memory organization. The minimal number of banks, being possibly a power of 2, proves the proposed organization advantageous with respect to the bandwidth utilization, the efficiency of the addressing realization, the bank selection, and the routing circuits.

In memory organizations with skewing schemes, the addressing and the bank selection calculations strongly depend on the number of the memory banks $\mathcal{B}$. This is because they require modulo and division operations with respect to $\mathcal{B}$. The proposed memory organization provides efficient solutions to the circuit design problem because it avoids the use of prime numbers, and moreover, because it allows $\mathcal{B}$ to be a power of 2. Regarding the bank selection, mapping $\Phi(x, y)$ requires only the following shift-add operations: left-shift x by $log_2 n$ bits, right-shift x by $log_2 m$ bits, and addition of these values to y. Since m and n are predefined and constant values, the resulting hardware is a small depth-two adder tree. Regarding the address calculations, many mappings can be applied to accommodate the proposed organization. For example, if we denote by M the image width and form a constant $const = \lceil \frac{M}{m} \rceil$, then a straightforward mapping is

$$addr(x, y) = \left\lfloor \frac{x}{m} \right\rfloor + \left\lfloor \frac{y}{n} \right\rfloor \cdot const. \tag{5}$$

Choosing the $const$ to be a power of 2 leads to the implementation of (5) with a single addition of the bit-shifted values of x and y.

In most applications, the data accessed (read) from the parallel memory must be rearranged to conform with a predefined order before they are loaded into the processing units (e.g., raster scan). Therefore, the outputs of the memory banks must be routed to the output of the memory by using permutation networks, which reorder the data according to the application specifications. In graphics applications, the number of required permutations increases with the number of supported access patterns and with the number of possible access locations on the image. Such routing circuits can be quite subtle to design or/and quite costly to implement in hardware. Focusing on improving the interconnection networks for graphics applications, this work has concentrated on exploiting the advantages of the linear skewing schemes. More specifically, the proposed mapping uses a simple composition of two linear skews: a linear column skew within each image stripe and a linear skew for the stripes (Section 3). Consequently, the organization maintains the *isotropic* property within each stripe and relatively at the stripe boundaries. Therefore, for interconnection, we can employ widely used designs such as the barrel shifters. Moreover, when $\mathcal{B}$ is a power of 2, the implementation of a $\lceil log_2 \mathcal{B} \rceil$ stage interconnection prevails over the complicated hardware realization imposed by the peculiarities of a prime $\mathcal{B}$.

To estimate the memory bandwidth utilization, we assume that the application performs correlated requests, i.e., scanning *MacroSquares*. We use a statistical analysis because the utilization depends on the origin of each request (Section 4). We assume a uniform distribution of the requests for *MacroSquares* on the image and we exclude the *Column* and the *Sparse-m* access formats because these formats lead unconditionally to 100 percent bandwidth utilization.

The scan of a *MacroSquare* originating at (x, y) on the image, with $x \bmod (m) = 0$, has no *local conflicts* for any selected access format. Such *MacroSquares* are requested with probability $\frac{1}{m}$ and result in 100 percent utilization of the memory. For the scan of any other *MacroSquare*, we use one extra cycle—as presented in Section 5—to avoid the *local conflicts* during the mn requests. The latter *MacroSquares* are requested with probability $\frac{m-1}{m}$ and their access leads to $\frac{mn}{mn+1}$ bandwidth utilization. Therefore, the average bandwidth utilization of the proposed memory organization when accessing arbitrary *MacroSquares* with any *basic* access format (besides *Column* and *Sparse-m*) is

$$AU_{MS} = \frac{m^2 n + 1}{m^2 n + m} = \frac{\mathcal{B} + 1/m}{\mathcal{B} + 1}.$$

This utilization is better than that derived from [3] by a factor of $\frac{\mathcal{B} + \frac{1}{m}}{\mathcal{B}}$.

8 Conclusion

This paper introduces a technique for designing efficient parallel memory organizations for graphics applications. The technique bases on a mapping Φ leading to a memory structure parameterized respectfully to the number of modules, allowing the number of modules to range from a power of 2 to any number and for any application. The results of this work prove that the efficiency of parallel accessing pixels in various patterns can be significantly improved by using the mapping Φ. These patterns, either contiguous or sparse, include the most often utilized shapes in image/video algorithms. Moreover, this work assumes a generic correlation among consecutive accesses in image/video algorithms in order to propose a strategy for avoiding the conflicts. The strategy retrieves the forthcoming conflicts by introducing the probability of utilizing at most one extra cycle every mn accesses depending on the access pattern used and its origin on the image.

The probability of using an extra cycle every mn access cycles, in the course of the algorithm, is an improved overhead comparing to the extra memory module(s) involved in related organizations. The simulation of graphics applications verifies that in certain cases, the performance of the proposed organization is equivalent to that of techniques utilizing greater number of banks. In other cases, the cycle overhead introduced by Φ is lesser than the bank overhead of the hitherto published techniques. The proposed design strategy leads to improved bandwidth utilization, and furthermore, due to the majority of graphics applications involving accesses of dimensions equal to powers of 2, it reduces the cost of realizing the bank/address computations and it provides a straightforward design for the routing network. The key advantage of the design technique is fulfilling the requirements of the graphics applications, though including a minimal number of memory modules—equal to the number of requested pixels.

Appendix

Proofs

Proof of Lemma 4.1. Using integer arithmetic properties [18], [19]:

$$\left\lfloor\frac{x+i\cdot s}{m}\right\rfloor - \left\lfloor\frac{x}{m}\right\rfloor = \left\lfloor\frac{i\cdot s}{m}\right\rfloor + \left\lfloor\frac{x \bmod (m) + (i\cdot s) \bmod (m)}{m}\right\rfloor = \left\lfloor\frac{i\cdot s}{m}\right\rfloor + \varepsilon_{xi},$$

where ε_{xi} is clearly either 0 or 1.

Further, using modulo properties [18], [19] and the fact that $s|m$ (i.e., $m = s\cdot m'$), we get the following:

$$\begin{aligned}
&\left(i\cdot s\cdot n + j\cdot s + \left\lfloor\frac{i\cdot s}{m}\right\rfloor + \varepsilon_{xi}\right) \bmod (mn)\\
&= \left(i\cdot s\cdot n + j\cdot s + \left\lfloor\frac{i\cdot s}{m}\right\rfloor + \varepsilon_{xi} - \left\lfloor\frac{i\cdot s}{m}\right\rfloor\cdot m\cdot n\right) \bmod (mn)\\
&= \left(i\cdot s\cdot n + j\cdot s + \left\lfloor\frac{i}{m'}\right\rfloor + \varepsilon_{xi} - \left\lfloor\frac{i}{m'}\right\rfloor\cdot m'\cdot s\cdot n\right) \bmod (mn)\\
&= \left(s\cdot n\cdot (i - \left\lfloor\frac{i}{m'}\right\rfloor\cdot m') + j\cdot s + \left\lfloor\frac{i}{m'}\right\rfloor + \varepsilon_{xi}\right) \bmod (mn)\\
&= \left(n\cdot s\cdot (i) \bmod (m') + j\cdot s + \left\lfloor\frac{i}{m'}\right\rfloor + \varepsilon_{xi}\right) \bmod (mn).
\end{aligned}$$

Therefore, congruence (2) can be written as

$$\Psi(i,j) + \varepsilon_{xi} \equiv 0 \pmod{mn}, \qquad (6)$$

where $\Psi(i,j) = (i) \bmod (m')\cdot \mathbf{ns} + j\cdot \mathbf{s} + \lfloor\frac{i}{m'}\rfloor$.

Case 1. $0 \le i < m$ and $0 \le j < n$.

In this case, the search for solutions (i,j) is constrained within $\mathbb{N}_m \times \mathbb{N}_n$, where $\mathbb{N}_l = [0, l-1]$. The mapping $\Psi(i,j)$: $\mathbb{N}_m \times \mathbb{N}_n \to \mathbb{N}_{mn}$ is isomorphic. To show its monomorphism, note that $\Psi(i,j)$ can be viewed as a finite mixed-radix numeral system [20] of the simple form

$$(\alpha_2\cdot\beta_1\beta_0 + \alpha_1\cdot\beta_0 + \alpha_0), \quad \begin{aligned}0 \le \alpha_0 \le \beta_0 - 1,\\ 0 \le \alpha_1 \le \beta_1 - 1,\end{aligned}$$

with $\beta_1 = n$, $\beta_0 = s$, $\alpha_2 = i \bmod (m')$, $\alpha_1 = j$, and $\alpha_0 = \lfloor\frac{i}{m'}\rfloor$. Each number is represented uniquely in such a system as $\alpha_2\alpha_1\alpha_0$. Further, note that each (i,j) corresponds to a distinct triplet $\alpha_2\alpha_1\alpha_0$ (each j corresponds to a distinct α_1 and each i corresponds to a distinct $\alpha_2\alpha_0$). Combining these two facts, we conclude that each (i,j) results in a distinct $\Psi(i,j)$. Regarding the epimorphism of $\Psi(i,j)$, it is straightforward to show that $0 \le \Psi(i,j) \le mn-1$. Therefore, Ψ maps $n\cdot m$ elements to a set with $n\cdot m$ elements. Since each mapping is distinct (as shown above), Ψ covers its entire range $\mathbb{N}_{mn}$.

Since $\varepsilon_{xi} \le 1$, we have $\Psi(i,j) + \varepsilon_{xi} \not> mn$ in $\mathbb{N}_m \times \mathbb{N}_n$, and thus, (6) has a solution only when $\Psi(i,j) + \varepsilon_{xi} = 0$, or when $\Psi(i,j) + \varepsilon_{xi} = mn$. Because $\Psi(i,j)$ is isomorphic, there is only one point for which $\Psi(i,j) = 0$. It is straightforward to show that this point is $(i,j) = (0,0)$ and that it is always a solution to (6). Further, iff $\varepsilon_{xi} = 1$ when Ψ obtains its maximum value, then there exists a second solution to (6). This solution must be an (i,j) for which $\Psi(i,j) = mn-1$. Because $\Psi(i,j)$ is isomorphic, there is only one such point: $(i,j) = (m-1, n-1)$.

Case 2. $0 \le i < m$ and $-n < j < 0$.

In this case, (6) has no solutions. It is straightforward to show that $-mn < \Psi(i,j) + \varepsilon_{xi} < mn$. Also, it can be proved, by contradiction, that $\Psi(i,j) + \varepsilon_{xi} \ne 0$: suppose that there exists a pair (i_r, j_r) that results in zero, and therefore,

$$ns\cdot(i_r \bmod m') + \left\lfloor\frac{i_r s}{m}\right\rfloor + \left\lfloor\frac{x \bmod m + (i_r s) \bmod m}{m}\right\rfloor = |j_r|s$$

$$\Rightarrow s < ns\cdot(i_r \bmod m') + \left\lfloor\frac{x \bmod m + i_r s}{m}\right\rfloor < ns,$$

which is an immediate result of the j constraint. Observe that the value $i_r \bmod m'$ must be zero, or else the expression will not evaluate to an integer less than ns. Consequently, i_r must be a multiple of m', i.e., $i_r = i'_r\cdot m'$. Substituting i_r to the above expression, we conclude that the existence of the (i_r, j_r) pair implies the correctness of the inequality $s < i'_r < ns$. This inequality cannot be true because $i_r < m$ (lemma hypothesis), and therefore, $i'_r < s$. □

Proof of Lemma 4.2. The difference from the proof of Lemma 4.1 lies in the fact that when $i = m-1$ and $x \bmod (m) < s$, we have

$$\varepsilon_{xi} = \left\lfloor\frac{x \bmod (m) + (i\cdot s) \bmod (m)}{m}\right\rfloor = \left\lfloor\frac{x \bmod (m) - s + m}{m}\right\rfloor = 0$$

because $x \bmod (m) - s + m < m$.

Since ε_{xi} is zero when $\Psi(i,j)$ obtains its maximum value $mn-1$, there can be no (i,j) such that $\Psi(i,j) + \varepsilon_{xi} = mn$. Therefore, the only solution to (2) is the one for which $\Psi(i,j) + \varepsilon_{xi} = 0$, i.e., $(i,j) = (0,0)$. □

Proof of Theorem 4.2. The pixels of an *s-s Subsampled* area $m \cdot s \times n \cdot s$ (i.e., a *Sparse-s* pattern) can be referenced using relative coordinates $i' = i \cdot s$ and $j' = j \cdot s$, where $i \in [0, m-1]$ and $j \in [0, n-1]$. From $\Phi(x_p, y_p) = \Phi(x_p + i', y_p + j')$, we derive (2). Because $s|m$, we can use Lemma 4.1 and the arguments used in the proof of Theorem 4.1 to conclude this proof. □

Proof of Theorem 4.3. Two pixels (x_p, y_p) and (x_q, y_q) within the arbitrary column reside in the same memory bank when $\Phi(x_p, y_p) = \Phi(x_q, y_q)$. Using relative coordinates, that is, when $\Phi(x_p, y_p) = \Phi(x_p, y_p + j)$, $j \in [1, mn-1]$. The derived congruence equation $j \equiv 0 \bmod mn$ has no solution when $0 < j < mn$, and thus, $\Phi(x_p, y_p) = \Phi(x_q, y_q)$ never holds. □

Proof of Theorem 4.4. When using relative coordinates for two pixels within an arbitrary row of length mn, from $\Phi(x_p, y_p) = \Phi(x_p + i', y_p)$, $i' \in [0, mn-1]$, we derive the following equation:

$$i' \cdot n + \left\lfloor \frac{x_p + i'}{m} \right\rfloor - \left\lfloor \frac{x_p}{m} \right\rfloor \equiv 0 \pmod{mn}.$$

Substituting i' with $j \cdot m + i$, where $j \in [0, n-1]$ and $i \in [0, m-1]$ (it is straightforward to show that this is an isomorphic mapping), we derive the equivalent congruence:

$$i \cdot n + j + \left\lfloor \frac{x_p + i}{m} \right\rfloor - \left\lfloor \frac{x_p}{m} \right\rfloor \equiv 0 \pmod{mn},$$

which is the equation studied in Lemma 4.1 with $s = 1$. Therefore, the proof of Theorem 4.4 is reduced to the proof of Theorem 4.1. Using the same arguments, the only pixel for which (2) might have more than one solution within the row is the origin pixel of the row. The second solution in this case cannot be other than $(i, j) = (m-1, n-1)$, corresponding to $i' = mn - 1$. Therefore, any arbitrary row of length $mn - 1$ can be conflict-free accessed. □

Proof of Theorem 4.5. The pixels of the *multisquare* can be referenced with coordinates $(x + i', y + j')$, where $i' = \lfloor \frac{i}{r} \rfloor r \cdot m + (i) \bmod (r)$ and $j' = \lfloor \frac{j}{r} \rfloor r \cdot n + (j) \bmod (r)$, with $i \in [0, n-1]$ and $j \in [0, m-1]$. To show that these pixels are stored in distinct memory banks, we show that the mapping $\Phi'(i,j) = \Phi(x + i', y + j') : \mathbb{N}_n \times \mathbb{N}_m \rightarrow \mathbb{N}_{nm}$ is isomorphic. Substituting i' and j' with $i' = i \cdot m - (i) \bmod (r) \cdot (m-1)$ and $j' = j \cdot n - (j) \bmod (r) \cdot (n-1)$ in Φ, we get (using integer arithmetic and modulo properties [18], [19]):

$$\begin{aligned}
&\Phi(x + i', y + j') \\
&= \left(\Xi(i,j) + y + xn + \left\lfloor \frac{x + i \bmod (r)}{m} \right\rfloor \right) \bmod (mn) \\
&= \left(\Xi(i,j) + \left\lfloor \frac{x \bmod (m) + i \bmod (r)}{m} \right\rfloor + \mathit{const} \right) \bmod (mn),
\end{aligned}$$

where

$$\Xi(i,j) = \left(i \bmod (r) + \left\lfloor \frac{j}{r} \right\rfloor r \right) \cdot n + \left(j \bmod (r) + \left\lfloor \frac{i}{r} \right\rfloor r \right).$$

Since $x \bmod (m) \leq m - r$, we have that

$$\left\lfloor \frac{x \bmod (m) + i \bmod (r)}{m} \right\rfloor = 0.$$

Therefore, to complete the proof, it suffices to show that the mapping $\Xi(i,j)$: $\mathbb{N}_n \times \mathbb{N}_m \rightarrow \mathbb{N}_{nm}$ is isomorphic. We use a similar argument with that used in the proof of Lemma 4.1; note that $\Xi(i,j)$ can be viewed as a finite numeral system of the form

$$(\alpha_1 \cdot \beta_0 + \alpha_0), \quad 0 \leq \alpha_0 \leq \beta_0 - 1,$$

with $\alpha_1 = (i \bmod (r) + \lfloor \frac{j}{r} \rfloor r)$ and $\alpha_0 = (j \bmod (r) + \lfloor \frac{i}{r} \rfloor r)$. Further, note that α_1 and α_0 can be viewed individually as numeral systems of the above form. It is straightforward to show that distinct (i,j) pairs result in distinct representations $\alpha_1 \alpha_0$ and that $0 \leq \Xi(i,j) \leq mn - 1$. It follows that $\Xi(i,j)$ is an isomorphic mapping, as is $\Phi'(i,j)$, and thus, Φ does not map two distinct pixels of *multisquare* in the same memory bank. □

Proof of Lemma 5.1. In cases with *local conflicts*, we assume that $\lfloor \frac{x \bmod (m)}{s} \rfloor \geq 1$ and the access format is not a *Column* (corollaries of Section 4). Also $s = 1$ for the *contiguous* patterns and $s|m$ in case of *sparse* patterns. To prove the lemma, we show that $mbank(x,y,s) \neq \Phi(x_q, y_q)$ for each pixel (x_q, y_q) of the format, which leads to the *local conflict*. To do so, we express the pixels of the format with coordinates (i,j) relative to (x,y) and show that the equation $\Phi(x+i, y+j) = mbank(x,y,s)$ has no solutions (i,j) pointing at a pixel of the format under examination. Following this procedure for any access format, we derive the generic congruence equation (for the *Row* format we use the coordinate transformation used in the proof of Theorem 4.4):

$$n \cdot \left\lfloor \frac{x \bmod (m)}{s} \right\rfloor \cdot s + \varepsilon_{xi} + \Psi(i,j) \equiv 0 \pmod{mn}, \qquad (7)$$

where $i \in [0, m-1], j \in [0, n-1]$, and $\Psi(i,j)$ is the isomorphic mapping, $\mathbb{N}_m \times \mathbb{N}_n \rightarrow \mathbb{N}_{mn}$, used in the proof of Lemma 4.1. The integer at the left-hand side of (7) cannot be greater than or equal to $2mn$ because $\Psi(i,j) < mn$, $\varepsilon_{xi} \leq 1$, and $\lfloor \frac{x \bmod (m)}{s} \rfloor \cdot s \leq m - 1$, and it cannot be zero because $\lfloor \frac{x \bmod (m)}{s} \rfloor \geq 1$ (lemma hypothesis). Therefore, (7) has a solution only when the above-mentioned integer equals mn, i.e., when $\Psi(i,j) + \varepsilon_{xi} = K$, where $K = n \cdot (m - \lfloor \frac{x \bmod (m)}{s} \rfloor s)$. We show that $\Psi(i,j) + \varepsilon_{xi} \neq K$ because when $\Psi(i_o, j_o) = K$, we have $\varepsilon_{xi_o} = 1$, and when $\Psi(i_u, j_u) = K - 1$, we have $\varepsilon_{xi_u} = 0$ (ε_{xi} can be either 0 or 1). Since $\Psi(i,j)$ is isomorphic, we have exactly one (i_o, j_o) and exactly one (i_u, j_u):

$$\begin{aligned}
(i_o, j_o) &= \left(m' - \left\lfloor \frac{x \bmod (m)}{s} \right\rfloor, \ 0 \right), \\
(i_u, j_u) &= \left(m - \left\lfloor \frac{x \bmod (m)}{s} \right\rfloor - 1, \ n - 1 \right).
\end{aligned}$$

In the first case, we get $\varepsilon_{xi_o} = \lfloor \frac{m + (x \bmod m) \bmod (s)}{m} \rfloor = 1$, and in the second case, $\varepsilon_{xi_u} = \lfloor \frac{m + (x \bmod m) \bmod (s) - s}{m} \rfloor = 0$. □

REFERENCES

[1] P. Budnick and D.J. Kuck, "Organization and Use of Parallel Memories," *IEEE Trans. Computers,* vol. 20, no. 12, pp. 1565-1569, Dec. 1971.

[2] D.C. VanVoorhis and T.H. Morrin, "Memory Systems for Image Processing," *IEEE Trans. Computers,* vol. 27, no. 2, pp. 113-125, Feb. 1978.

[3] B. Chor, C.E. Leiserson, R.L. Rivest, and J.B. Shearer, "An Application of Number Theory to the Organization of Raster-Graphics Memory," *J. ACM,* vol. 33, no. 1, pp. 86-104, Jan. 1986.

[4] K. Kim and V.K. Prasanna, "Latin Squares for Parallel Array Access," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 4, pp. 361-370, Apr. 1993.

[5] A. Deb, "Multiskewing—a Novel Technique for Optimal Parallel Memory Access," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 6, pp. 595-604, June 1996.

[6] J.W. Park, "Multiaccess Memory System for Attached SIMD Computer," *IEEE Trans. Computers,* vol. 53, no. 4, pp. 439-452, Apr. 2004.

[7] J.K. Tanskanen, R. Creutzburg, and J.T. Niittylahti, "On Design of Parallel Memory Access Schemes for Video Coding," *J. VLSI Signal Processing Systems,* vol. 40, no. 2, pp. 215-237, June 2005.

[8] R.F. Sproull, I.E. Sutherland, A. Thompson, S. Gupta, and C. Minter, "The 8 by 8 Display," *ACM Trans. Graphics,* vol. 2, no. 1, pp. 32-56, Jan. 1983.

[9] J.M. Frailong, W. Jalby, and J. Lenfant, "XOR-Schemes: A Flexible Data Organization in Parallel Memories," *Proc. 1985 Int'l Conf. Parallel Processing,* pp. 276-283, Aug. 1985.

[10] H. Vandierendonck and K. De Bosschere, "XOR-Based Hash Functions," *IEEE Trans. Computers,* vol. 54, no. 7, pp. 800-812, July 2005.

[11] C. Liu, X. Yan, and X. Qin, "An Optimized Linear Skewing Interleave Scheme for On-Chip Multi-Access Memory Systems," *Proc. 17th Great Lakes Symp. VLSI,* pp. 8-13, Mar. 2007.

[12] J. Tanskanen, T. Sihvo, J. Niittylahti, J. Takala, and R. Creutzburg, "Parallel Memory Access Schemes for H.263 Encoder," *Proc. IEEE Int'l Symp. Circuits and Systems,* vol. 1, pp. 691-694, May 2000.

[13] R. Raghavan and J.P. Hayes, "On Randomly Interleaved Memories," *Proc. 1990 ACM/IEEE Conf. Supercomputing,* pp. 49-58, Nov. 1990.

[14] N. Topham and A. Gonzalez, "Randomized Cache Placement for Eliminating Conflicts," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 185-192, Feb. 1999.

[15] A. Vitkovski, G. Kuzmanov, and G. Gaydadjiev, "Memory Organization with Multi-Pattern Parallel Accesses," *Proc. Conf. Design, Automation and Test in Europe,* pp. 1414-1419, Mar. 2008.

[16] K. Babionitakis, G. Lentaris, K. Nakos, N. Vlassopoulos, D. Reisis, G. Doumenis, G. Georgakarakos, and I. Sifnaios, "A Real-Time Motion Estimation FPGA Architecture," *J. Real-Time Image Processing,* vol. 3, nos. 1/2, pp. 3-20, Mar. 2008.

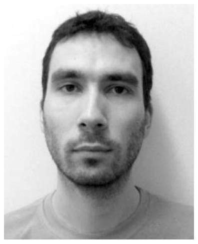
[17] "H.264 Advanced Video Coding for Generic Audiovisual Services,"ITU-T, May 2003.

[18] T.M. Apostol, *Introduction to Analytic Number Theory.* Springer-Verlag, 1976.

[19] V. Shoup, *A Computational Introduction to Number Theory and Algebra.* Cambridge Univ. Press, 2005.

[20] D.E. Knuth, *Art of Computer Programming, Volume 2: Seminumerical Algorithms,* chapter 4. Addison-Wesley, 1981.

[21] R.C. Gonzalez and R.E. Woods, *Digital Image Processing.* Prentice-Hall, 2002.



George Lentaris received the degree in physics and the MSc degree in electronic automation from the National Kapodistrian University of Athens (NKUA) in 2004 and 2006, respectively. He is currently working toward the PhD degree in the Electronics Laboratory at the Department of Physics, NKUA. His research interests include algorithms and parallel architectures for applications in signal processing and image/video processing.

Dionysios Reisis received the ptychion degree in electrical engineering from the University of Patras, Greece, in 1983, and the MSc and PhD degrees in computer engineering from the Department of Electrical and Computer Engineering at the University of Southern California in 1989. In 1990, he joined the Telecommunications Lab of the Division of Computer Science at the National Technical University of Athens (NTUA) as a research associate. In 1991, he became a lecturer. Since 1996, he has been an assistant professor in the Electronics Laboratory of the Department of Physics at the University of Athens (NKUA). His research interests include parallel architectures and algorithms for image and graph signal processing with applications in VLSI environment, as well as real-time hardware design and efficient algorithms design for telecommunication systems support.