Memory Partitioning for Multidimensional Arrays in High-level Synthesis

Yuxin Wang,[1] Peng Li,[1] Peng Zhang,[2] Chen Zhang,[1] Jason Cong[1,2,3]
[1] Center for Energy-Efficient Computing and Applications, Computer Science Department, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]UCLA/PKU Joint Research Institute in Science and Engineering
{ayerwang, peng.li, chen.ceca}@pku.edu.cn, {pengzh, cong }@cs.ucla.edu

ABSTRACT

Memory partitioning is widely adopted to efficiently increase the memory bandwidth by using multiple memory banks and reducing data access conflict. Previous methods for memory partitioning mainly focused on one-dimensional arrays. As a consequence, designers must flatten a multidimensional array to fit those methodologies. In this work we propose an automatic memory partitioning scheme for multidimensional arrays based on linear transformation to provide high data throughput of on-chip memories for the loop pipelining in high-level synthesis. An optimal solution based on Ehrhart points counting is presented, and a heuristic solution based on memory padding is proposed to achieve a near optimal solution with a small logic overhead. Compared to the previous one-dimensional partitioning work, the experimental results show that our approach saves up to 21% of block RAMs, 19% in slices, and 46% in DSPs.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Design Aids–*automatic synthesis*

General Terms

Algorithms, Performance, Design

Keywords

High-Level Synthesis, Memory Partitioning, Memory Padding

1. INTRODUCTION

To balance the requirements of high performance, low power and short time-to-market, field programmable gate array (FPGA) devices have gained a growing market against ASICs and general-purpose processors over the past two decades. Recently, FPGAs have also been used as general computing platforms as alternatives to CPUs and GPUs. Although FPGAs provide plenty computational units for parallelization, how to supply those units with the required high-speed data streams is a major challenge.

This is especially true after loop unrolling and pipelining, when multiple data elements from the same array are often required simultaneously in a single clock cycle. Typical on-chip block RAMs (BRAMs) in FPGAs have two access ports. A straightforward solution is to duplicate the array into multiple copies [13]. Although the duplication approach can support simultaneous read operations, it may have significant area and power overhead and introduce memory consistency problem. A better approach is to partition the original array into multiple memory banks. Each bank holds a portion of the original data and serves a limited number of memory requests.

Memory partitioning has been studied in the distributed computing domain for decades [8, 15], where data elements are partitioned into different processors to reduce communication among the processors. While some of the partitioning algorithms in distributed computing can be directly applied to high-level synthesis, the freedom of creating memory banks tailored to the target application can lead to more efficient memory partitioning algorithms for high-level synthesis [19, 3, 6, 20, 12]. In [19], different fields of a single *structure* are partitioned into multiple memory banks for data parallelism based on profiling results. In [3], a single array is decomposed into disjoint memory banks for storage minimization purposes through accurate lifetime analysis using a polyhedral model. The purpose of the memory partitioning algorithm presented in this paper is to improve system performance by assigning memory accesses to disjoint memory banks and providing simultaneous conflict-free memory accesses [6, 20, 12], which is orthogonal to the problem in [3]. In [6], an automated memory partition algorithm is proposed to support multiple simultaneous affine memory references to the same array. The algorithm can be extended to efficiently support memory references with modulo operations (common after data reuse using scratchpad memory) with limited memory paddings [20]. In [12], memory accesses in different loop iterations can be partitioned into different memory banks and scheduled into the same cycle to minimize the number of required memory banks.

However, previous memory partitioning algorithms are designed for one-dimensional arrays, while many designs for FPGAs are often specified by nested loops with multidimensional arrays—such as image, video, and scientific computing applications. In previous works, a multidimensional array is first flatted into a single-dimensional array before memory partitioning. However, memory addresses after array flattening are dependent on the array size. For different array sizes, different partitioning schemes are generated, many of which are suboptimal. In this paper we focus on providing an effective and efficient memory partition algorithm for multidimensional arrays based on linear transformation.

The main contributions of this work are described as follows:

1) A linear-transformation-based multidimensional memory partition algorithm is proposed to generate the smallest memory bank numbers regardless of the size of input array.
2) An optimal inner-bank offset generation scheme is proposed based on point counting in polytopes.

3) A heuristic solution based on memory padding is proposed to achieve a near-optimal inner-bank offset generation with a comparative small logic overhead and storage overhead.

The remainder of this paper is organized as follows: Section 2 provides a motivational example for the multidimensional memory partitioning; Section 3 formulates the problem, and Section 4 describes the detailed solution; Section 5 analyzes the experimental results, and is followed by conclusions in Section 6.

2. MOTIVATIONAL EXAMPLE

Our motivational example, as shown in Fig. 1(a), is from a loop kernel of the 2D denoise algorithm, which is a key application in medical image processing [2]. The kernel has five accesses to the array *A* in the inner loop. Fig. 1(c) shows the access pattern of the inner loop iteration and the partition based on linear transformation, where x_0 is the lower-dimension index and x_1 denotes the index in higher dimension. The light points in Fig. 1(c) represent the data elements in the array with the dark points representing the elements accessed in a single loop iteration. We assume that the physical memory has one read port—only one data element can be read from a physical memory in each clock cycle. To improve the processing throughput of the loop kernel, we need to pipeline the execution of successive inner loop iterations, which means that multiple accesses to the same array will happen in one clock cycle. If array elements are not properly allocated in multiple physical memory banks, memory conflicts will occur and pipeline performance will be impacted.

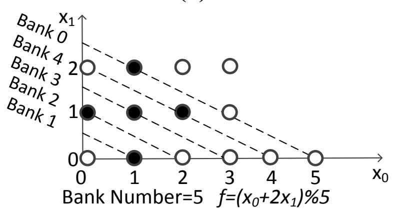
Previous memory partitioning solutions mainly focus on 1-D arrays, as in [6]. It flattens the array first, as shown in Fig. 1(b), and then partitions the flattened array. In order to fully pipeline the loop, five elements of data are required in each clock cycle. Thus the minimum number of memory banks for a non-conflict partitioning is five. However a cyclic partition with five banks can not satisfy the non-conflict constraint according to the code in Fig. 1(b). Take iteration (i, j)=(1, 1) for example, the second reference (A[64*j+i-1]=A[64]) and the forth reference (A[64*j+i+64]=A[129]) will access the same bank (64%5 = 129%5). Using the approach in [6] on the flattened array, we can prove that at least six banks are required.

```
int A[64][64];
for j= 1 to 62
  for i = 1 to 62
      b[j][i] = f(A[j][i], A[j][i-1], A[j-1][i ], A[j+1][i], A[j][i+1]);
    //accesses to down, up, left, and right
```

(a)

```
int A[4096];
for j= 1 to 62
  for i = 1 to 62
     b[j][i] = f(A[64*j+i], A[64*j+i-1], A[64*j+i-64], A[64*j+i+64],
                 A[64*j+i +1]);
```

(b)



(c)

Fig. 1 Denoise: (a) original loop kernel, (b) loop kernel with flattened array, (c) multidimensional partitioning based on linear transformation

In fact, using the linear transformation based multidimensional partitioning method proposed in this work, the original code (Fig. 1(a)) can be fully pipelined with five memory banks. As illustrated in Fig. 1(c), the data elements on the same dotted line will be partitioned into the same memory bank, e.g., the data A[0][2] and A[1][0] are in the same bank. Whereas, the five data elements accessed in one inner-loop iteration are mapped into five different banks; i.e., in iteration (i, j)=(1, 1), the second reference (A[1][0]) and the forth reference (A[2][1]) are no longer in the same bank. Based on the linear transformation method we proposed, the code in Fig. 1(a) is partitioned with a linear transformation $f = (x_0 + 2x_1)\%5$ (as shown in Fig. 1(c)). We will describe the detailed partitioning algorithm in Sections 3 and 4.

3. PROBLEM FORMULATION

In this paper we will describe how we partition several multidimensional memory references in a multidimensional loop nest to separate memory banks to enable loop pipelining with simultaneous memory accesses. For simplicity, loop initiation interval (II) and physical memory port number are both assumed to be 1 in this paper. Algorithms and formulations can be extended for any constant loop initiation interval and physical memory port number by scheduling and mapping the accesses onto different time intervals and physical memory ports (as presented in [6]).

DEFINITION 1 (ITERATION DOMAIN [10]) Given a l-level loop nest with the iteration variables $i_0, i_1, \dots, i_{l-1}$ from outermost to innermost loop, the iteration vector is a vector of iteration variables, $\vec{\imath} = (i_0, i_1, \dots, i_{l-1})^T$. The *iteration domain D* is a set of all iteration vectors in the loop bounds.

DEFINITION 2 (AFFINE MEMORY REFERENCE) Given a d-dimensional array, a d-dimensional *affine memory reference* to the array is a set of linear combinations of iteration vectors and a constant:

$$R = \begin{pmatrix} a_{0,0} & \cdots & a_{0,l} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \cdots & a_{d-1,l} \end{pmatrix} * (i_0, i_1, \dots, i_{l-1}, 1)^T$$

where $a_{k,j} \in \mathbb{Z}$ is the coefficient of the j-th iteration vector in the k-th dimension.

DEFINITION 3 (DATA DOMAIN) Given a loop with m affine memory references $R_0, R_1, \dots, R_{m-1}$ on the same array, the *data domain M* of the array is defined as a set of all memory elements accessed by any memory reference in any loop iteration. Assuming the memory element accessed by memory reference R_j in iteration $\vec{\imath}$ is represented as $R_j(\vec{\imath})$, then

$$M = \bigcup_{\vec{\imath} \in D; 0 \le j < m} R_j(\vec{\imath})$$

DEFINITION 4 (MEMORY PARTITION) A *memory partition* of an array with data domain M is described as a pair of mapping functions $(f(d), g(d))$, $\forall d \in M$, where $f(d)$ is the bank number that d is mapped to, and $g(d)$ is the corresponding inner bank offset. Also $f(d) \ge 0$, and $g(d) \ge 0$.

After memory partitioning, a data element in the original array is allocated on a new memory bank with a new array offset (inner bank offset). The validation of the partitioning is interpreted as two distinct data elements mapped onto either different memory banks or the same bank with different inner bank offsets. A valid memory partition of an array with data domain M is described as $\forall d_1, d_2 \in M$,

$$d_1 \ne d_2 \iff (f(d_1), g(d_1)) \ne (f(d_2), g(d_2))$$

where $(f(d_1), g(d_1)) \neq (f(d_2), g(d_2))$ means

$$f(d_1) \neq f(d_2) \qquad \text{or} \qquad f(d_1)=f(d_2), g(d_1) \neq g(d_2)$$

An access conflict between two memory references R_j and R_k $(0 \leq j < k < m)$ means that $\exists \vec{i} \in D,\ R_j(\vec{i}), R_k(\vec{i}) \in M$

$$f(R_j(\vec{i}))=f(R_k(\vec{i}))$$

This access conflict constraint is under the assumption that each physical memory only has one port. With the preceding definitions and formulations, we use Problem 1 defined below to formulate the multidimensional memory partitioning problem. Eqn. (1) defines the optimality of memory partitioning, as our main objective is to minimize the memory bank number. Eqn. (2) is responsible for the validity of the partitioning. Eqn. (3) ensures no conflict access in any iteration, which is required for fully-pipelined loops.

PROBLEM 1. (BANK NUMBER MINIMIZATION). Given a loop with *m* affine memory references $R_0, R_1, \ldots, R_{m-1}$ on the same array, find the optimal memory partition *f*, such that:

$$\begin{aligned} &\textit{Minimize} \quad bank\_num = \max_{0 \leq i < m}\{f(R_i)\} && (1) \\ &\textit{Subject to} \quad \forall d_1, d_2 \in M,\ (f(d_1), g(d_1)) \neq (f(d_2), g(d_2)) && (2) \\ &\qquad \forall \vec{i} \in D, 0 \leq j < k < m, f(R_j(\vec{i})) \neq f(R_k(\vec{i})) && (3) \end{aligned}$$

The storage overhead minimization problem is formulated as Problem 2 under the same valid partition and non-conflict constraints as Problem 1.

PROBLEM 2 (STORAGE MINIMIZATION). Given a loop with *m* affine memory references $R_0, R_1, \ldots, R_{m-1}$ on the same array, a memory partition number *N*, find the inner bank offset function *g and* check globally for consistency such that:

$$\begin{aligned} &\textit{Minimize storage} = \textstyle\sum_{n=0}^{N-1} G_n \\ &\textit{Subject to} \quad 0 \leq n < N,\ G_n = \max_{0 \leq i < m, f(R_i)=n} g(R_i) \\ &\qquad 0 \leq i < m, f(R_i) \leq N \\ &\qquad \forall d_1, d_2 \in M,\ (f(d_1), g(d_1)) \neq (f(d_2), g(d_2)) \\ &\qquad \forall \vec{i} \in D, 0 \leq j < k < m, f(R_j(\vec{i})) \neq f(R_k(\vec{i})) \end{aligned}$$

4. PARTITIONING ALGORITHM

In this paper, we propose a Linear Transformation Based (LTB) memory partitioning algorithm. The algorithm is general enough to cover the solutions from previous array flattening based approaches. We only consider cyclic partitioning strategy in this work. Other partitioning schemes (as block and block-cyclic) can be applied based on this solution.

A *d*-dimensional memory index $\vec{x} = (x_0, x_1, \ldots, x_{d-1})^T$ is first transformed by $\vec{x} \rightarrow \vec{\alpha} * \vec{x}$, where $\vec{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{d-1})$, $\alpha_i \in \mathbb{Z}$. According to the properties of cyclic partitioning, the bank mapping function *f* is described as

$$bank: f(\vec{x}) = (\vec{\alpha} * \vec{x})\ \%\ N.$$

From a geometrical point of view, $\vec{\alpha} * \vec{x} = \mathrm{c}$ represents a series of hyperplanes in the data domain, where $\mathrm{c} \in \mathbb{Z}$, and $f(\vec{x})$ assigns the hyperplanes to different banks according to the value of *c%N*. The traditional array flattening approach is just a special case of LTB when $\vec{\alpha}$ is decided by the dimensional width, as shown in Example 1.

EXAMPLE 1 (Flattening Partition) Supposing that the dimensional width of the target array from low dimension to high dimension is $w_0, \ldots, w_{d-1}$, the traditional approach will first flatten the reference into one dimension. Then the array is cyclically partitioned, using modulo and division operations to generate the bank number and inner bank offset. The bank mapping function f and inner bank offset function g are described as below.

$$f(\vec{x}) = (x_{d-1} * \textstyle\prod_{k=0}^{d-2} w_k + x_{d-2} * \prod_{k=0}^{d-3} w_k + \ldots + x_1 * w_0 + x_0)\%N$$

$$g(\vec{x}) = (x_{d-1} * \textstyle\prod_{k=0}^{d-2} w_k + x_{d-2} * \prod_{k=0}^{d-3} w_k + \ldots + x_1 * w_0 + x_0)/N$$

We can see that the flattening partition is just a special case in LTB method with the coefficient $\vec{\alpha}$ equal to $(1, w_0, w_1, \ldots, \prod_{k=0}^{d-3} w_k, \prod_{k=0}^{d-2} w_k)$.

4.1 Bank Mapping

Extending the constraint provided by work in [6], we build our own non-conflict constraint for *d*-dimensional array references as Theorem 1. It offers a sufficient condition for the conflict-free accesses regulated by Eqn. (3). With the constraint, we can find the candidate linear transformation vectors that meets the requirement. Assuming that there are two *d*-dimensional array references as

$$R_0 = \begin{pmatrix} a_{0,0} & \cdots & a_{0,l} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \cdots & a_{d-1,l} \end{pmatrix} * (i_0, i_1, \ldots, i_{l-1}, 1)^T \text{ and}$$

$$R_1 = \begin{pmatrix} b_{0,0} & \cdots & b_{0,l} \\ \vdots & \ddots & \vdots \\ b_{d-1,0} & \cdots & b_{d-1,l} \end{pmatrix} * (i_0, i_1, \ldots, i_{l-1}, 1)^T,$$

the bank mapping for R_0 and R_1 with a linear transformation vector $\vec{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{d-1})$ is

$$f(R_0) = (\vec{\alpha} * R_0)\ \%\ N \quad \text{and} \quad f(R_1) = (\vec{\alpha} * R_1)\ \%\ N.$$

THEOREM 1. *Assuming that a d-dimensional array is accessed by two references* R_0 *and* R_1 *in an l-level loop nest, the array is cyclically partitioned into N banks with a linear transformation vector* $\vec{\alpha}$ *and a bank mapping function* f *so that the simultaneous accesses are not in conflict in the iteration domain, if*

$$gcd(\vec{\alpha} * \Delta_0{}^T, \vec{\alpha} * \Delta_1{}^T, \ldots, \vec{\alpha} * \Delta_{l-1}{}^T, N) \nmid \vec{\alpha} * \Delta_l{}^T \qquad (4)$$

where

$$\Delta_k = (\Delta_{0,k}, \Delta_{1,k}, \ldots \Delta_{d-1,k}), \Delta_l = (-\Delta_{0,l}, -\Delta_{1,l}, \ldots -\Delta_{d-1,l}),$$

$$\Delta_{j,k} = a_{j,k} - b_{j,k}, \forall 0 \leq k < l,\ 0 \leq j < d,$$

The detailed proof is in Appendix.

EXAMPLE 2. For a two dimensional array A[64][64] with two array references A[j][i], and A[j+1][i+1] in the inner loop iteration, the linear transformation vector $(\alpha_0,\alpha_1)=(1,2)$ and N=2 meets the non-conflict constraint according to $\gcd(0,0,2)=2 \nmid (1+2)$.

The candidate $\vec{\alpha}$ can be generated by exhaustive enumeration. We can use some constraints to reduce the searching space. First, it is obvious that $gcd(\alpha_0, \alpha_1, \ldots, \alpha_{d-1}) = 1$. Second, the optimal partition number is the number of the references *m*. For this target *N*, the searching space for the $\vec{\alpha}$ is N^d $(\forall 0 \leq j < d, 0 \leq \alpha_j < N)$. If $\vec{\alpha}$ is a candidate, for $\forall\, k \in \mathbb{Z}$ and $\forall 0 \leq j < d$, $\alpha_{j\_k} = k * N + \alpha_j$, $\vec{\alpha}_k = (\alpha_{0\_k}, \alpha_{1\_k}, \ldots, \alpha_{d-1\_k})$ also meets the constraint. In addition, the theorem can be easily extended to multiple references by detecting the conflict between each pair of references.

4.2 Constructing Inner Bank Offset Functions

Using techniques in Section 4.1, the candidate linear transformation vectors can be generated. In this section, we will specify how to calculate the inner bank offset for a given linear transformation vector. The principle is to keep the validation of the partitioning, which is that two different data can't be mapped to the same physical location. Our goal is to optimize Problem 2, for with different mapping functions, some physical locations may be mapped without any data so that an extra storage overhead is induced. Two approaches are introduced in this section.

4.2.1 Optimal Approach

An optimal approach to generate the inner bank offset is to scan the data in sequence. Since all of the data elements on the same hyperplane set ($(\alpha * x)\%N = c$) are in the same bank, scanning the data along the hyperplane set in sequence and use the sequence number as the inner bank offset can generate a valid memory partition without any extra storage overhead. The problem can be converted by integer point counting in a polytope using Ehrhart polynomial [9]. Two polytopes (a base polytope and an offset polytope) are formulated for a given point $x' = (x'_0, x'_1, \dots, x'_{d-1})$. Then the sum of the point number in the two polytopes is used as the inner bank offset for the point. We illustrate this process in Example 3. The detailed formulation and theory of integer point counting using Ehrhart polynomial is given in Appendix.

EXAMPLE 3. Given a candidate vector $\vec{\alpha}$=(1,2), the hyperplanes are described as $x_0 + 2x_1 = c$. For a given point x'= (3,1), the two polytopes are formed as in Fig. 2, in which the base polytope contains the hyperplanes with $c < 5$, and the offset polytope is on c=5. According to the theory in [9], the point numbers in the two polytopes are the functions of c and x'_0 separately. By using the Ehrhart tool in Polylib [21], we get the Ehrhart polynomials for each polytope as L_{base} and L_{offset}.

$$L_{base}(c) = \frac{1}{20} \times c^2 + \left(-\frac{1}{10} \times c\right) + [0, \frac{1}{20}, 0, -\frac{3}{20}, -\frac{2}{5}, \frac{1}{4}, -\frac{1}{5}, \frac{1}{4}, -\frac{2}{5}, -\frac{3}{20}]_c$$

$$L_{offset}(x'_0) = \frac{1}{2} \times x'_0 + [1, \frac{1}{2}]_{x'_0},$$

where $[u_0, u_1, \dots u_{p-1}]_c = u_l$ when $c\%p = l, p \in Z$.

When $x'=(x'_0, x'_1)$ = (3,1), c=5,

$L_{base}(5) = \frac{25}{20} - \frac{5}{10} + \frac{1}{4} = 1$, $L_{offset} = 2$,

$g(3,1) = L_{base} + L_{offset} = 3$

Using Ehrhart's point-counting method, we have the optimal solution to Problem 2, but we find that the area required for computing the optimal *g* fuctions can be very large. In Example 2, four multiplications and two tables (generating the constants in the end of the polynomials) are used. Although we can get the optimal solution to both Problem 1 and Problem 2, compared to the straightforward array-flattening method, the complex address generation in this method makes it not worth doing. As a result a trade-off between practicality and optimality is considered using a heuristic approach presented next.

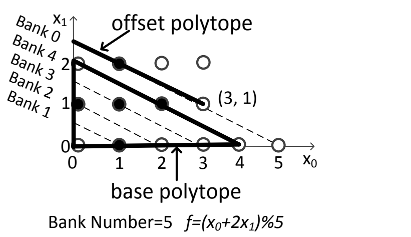


Fig. 2 An example of Ehrhart's point-counting

4.2.2 Heuristic Approach

Our heuristic method to efficiently find a linear transformation vector with a comparative simple inner bank offset function *g* is to do memory padding in the data domain. As stated before, a linear transformation vector $\vec{\alpha}$ for flattening partition is

$$\vec{\alpha} = (1, w_0, w_1, \dots, \prod_{k=0}^{d-3} w_k, \prod_{k=0}^{d-2} w_k).$$

It may lead to suboptimal partitioning as we depicted in the motivational example. Our memory padding method finds the coefficient vector $\vec{\alpha}$ with the validity guaranteed based on this given vector. Firstly, a padding vector $\vec{q} = (q_0, q_1, \dots, q_{d-1})$ is introduced, in which q_k represents the increase of size in *k*-th dimension. For a sub-domain formed by dimension *j* and dimension k $(0 \le j < k < d)$ with a given bank number *N* and bank linear transformation vector $\vec{\alpha} = (\dots \alpha_j, \dots, \alpha_k \dots)$, a padding size q_k, a valid partition should satisfy

$$\alpha_k(w_k + q_k) \bmod \mathrm{N} = \alpha_j \bmod \mathrm{N}.$$

It is equal to Eqn. (5).

$$\mathrm{N} | \alpha_k(w_k + q_k) - \alpha_j \tag{5}$$

The new linear transformation vector $\vec{\alpha}$ is

$$\vec{\alpha} = (1, wp_0, wp_1, \dots, \prod_{k=0}^{d-3} wp_k, \prod_{k=0}^{d-2} wp_k),$$

where $0 \le k < d, wp_k = w_k + q_k$.

The geometric meaning of the memory padding is that as each hyperplane only has one data element with the vector based on array flattening, the address is actually generated by scanning along a certain dimension. With a certain bank number, the allocation of the banks needs to be continuous between the last data element in the previous line and the first data element in the next line so that the partitioning validity is met. Fig. 3 shows an example for our method. To meet the validity of the partitioning with an optimal bank number five, dimension x_0 is increased by 2 (size increased from 64 to 66).

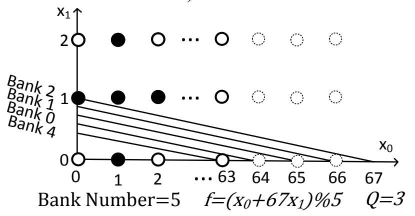


Fig. 3 An example for memory padding

The above discussion is based on a fixed dimension scanning order. But in fact the value of the element in $\vec{\alpha}$ implies the scanning order. For example, $\alpha_k = 1$ implies the scanning starts from dimension k. Thus when we change the value of $\vec{\alpha}$ in a range, the scanning order of the array and the total padding size will be changed. Through this, we minimize the extra storage overhead induced by memory padding. The padding size on each dimension is bounded within *N*, as each dimension is cyclically partitioned according to bank mapping function *f*. Eqn. (5) could be simplified as $\mathrm{N} | w_k + q_k$, then we have

$$q_k = \mathrm{N} * \left\lceil \frac{w_k}{N} \right\rceil - w_k \tag{6}$$

The maximum padding size on a *d*-dimensional array with N partition banks is calculated as

$$(N-1) * \frac{\prod_{k=0}^{d-1} w_k}{w_0} + (N-1) * \frac{\prod_{k=0}^{d-1} w_k}{w_1} +, \dots, +(N-1) * \frac{\prod_{k=0}^{d-1} w_k}{w_{d-2}}$$

4.3 Overall Flow

This section describes the overall flow while using memory padding based heuristic method. As the interplay between the padding size and the bank number, we give our flow to find the tradeoff between the optimal partition and extra storage overhead. The lower bound for the bank number N is the reference number m in the inner loop iteration. First, we fix the bank number N ($N \geq m$). Second, we find the possible padding $\vec{q}=(q_0, q_1, \dots, q_{d-1})$ under various array dimension orders with linear transformation coefficient $\vec{\alpha}=(\alpha_0, \alpha_1, \dots, \alpha_{d-1})$. And we'll get the best candidate vector $\vec{\alpha}_p$ with the total padding size minimized. Then we check whether $\vec{\alpha}_p$ satisfy the conflict-free constraint in Eqn. (4) with the bank number N. The detailed LTB algorithm is described as follows.

Step 1: Give the partition bank number $N = m$.
Step 2: Find every possible $\vec{\alpha}$ $(\forall 0 \leq j < d, 0 \leq \alpha_j < N)$ with a padding vector $\vec{q}$ according to Eqn. (6). Queue all the $\vec{\alpha}$ by the increase of the total padding size. Find $\vec{\alpha}_p$ with the minimum padding size.
Step 3: Check if $\vec{\alpha}_p$ meets the conflict-free requirement according to Eqn. (4). If $\vec{\alpha}_p$ cannot meet the requirement, find the next solution in the queue and recheck the conflict-free constraint.
Step 4: If there is no solution for *N*, *N=N+1*, go back to Step 2.

The complexity of searching for an array dimension order is $\prod_{k=1}^{d} k$. And according to Eqn. (6), we can actually calculate the padding size based on a given dimension order. This flow is capable to find a solution for both Problem 1 and Problem 2. It is optimal in Problem 1, and it provides a near optimal solution to Problem 2 with a bounded maximum extra storage overhead and a low complexity. Our experiments prove that in some cases the padding method can find an optimal solution and in other cases the gap between it with optimality is small.

5. EXPERIMENTAL RESULTS

5.1 Experiment Setup

The automatic multidimensional memory-partitioning flow is implemented in C based on the open source compiler infrastructure ROSE [14]. ROSE is a flexible translator supporting source-to-source code transformation. We use Vivado from Xilinx [17] as the high-level synthesis tool. The RTL output is implemented by Xilinx ISE 13.1 [18] on the target FPGA platform Xilinx Virtex-6. The implementation flow is illustrated in Fig. 4.

The high-level abstraction is parsed into the flow with the partition directives and constraints, such as target II. After memory partitioning analysis and source-to-source code transformation, the transformed code is synthesized by the high-level synthesis tool and followed by logic synthesis.

Six loop kernels are selected from the real applications as the benchmarks. As we focus on the effects brought by different access patterns, several of the benchmarks are the loop kernels from the same application with different access patterns. DENOISE_1 and DENOISE_2 are from the Rician-denoise algorithm [11] in medical image applications. DENOISE_1 is the original access pattern which accesses five data elements in the inner-loop iteration. DENOISE_2 is the access pattern by unrolling the loop in DENOISE_1 by 2. MOTION_LV and MOTION_C are the different loop kernels of motion compensation from official H.264 decoder JM 14.0 [4]. MOTION_LH is the motion compensation for luma samples in the video frame in the vertical direction, and MOTION_C is the interpolation for the chroma components. BICUBIC_INTER [1] is from bicubic interpolation process. And SOBEL [16] is from Sobel edge detection algorithm. (The detailed access patterns of the benchmarks are illustrated in Appendix)

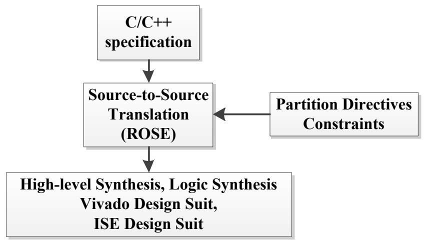

C/C++ specification
Source-to-Source Translation (ROSE)
Partition Directives Constraints
High-level Synthesis, Logic Synthesis
Vivado Design Suit,
ISE Design Suit

Fig. 4 Implementation flow

5.2 Experimental Results

The detailed experimental results are shown in Table 1, Table 2 and Table 3. We compared the experimental results for the state-of-art 1-dimensional partition algorithm with flattened arrays and our proposed linear transformation based algorithm (LTB). Table 1 shows the percentage of extra storage overhead when applying different memory size on DENOISE_1. The results after source-to-source transformation are shown in Table 2. And Table 3 shows the results after synthesizing. The algorithm from [6] for the flattening memory partitioning is re-implemented for comparison. As shown in Table 2, we list the *original II* of the pipelined loop and the *target II*. Our target throughput is *II=1*. The partitioning in both of the methods can meet the throughput requirement. The bank number for achieving the target throughput by using the flattening method and LTB are represented in the next two columns, followed by the essential *padding size* after applying LTB. Physical resource usage (block RAMs, slices, and DSPs) and timing information are reported by Xilinx ISE, and power estimation is given by Xilinx XPower Analyzer. The block RAMs are dual-port in the Xilinx Virtex-6.

Table 1 represents the percentage of padding size compared to the original array size (also the optimal solution). 140 different array sizes are applied in the experiments. And we found that the padding size is related tiny. In Table 2, we can see that our proposed LTB method improves the partitioning bank number on all of the six benchmarks. And five benchmarks have extra padding by using our LTB approach (DENOISE_1, DENOISE_2, MOTION_C, BICUBIC_INTER, and SOBEL). As each piece of the partition is relatively not too large and can fit in a BRAM, the padding size is totally negliable. However, if the arrays in the benchmarks are originally large, memory padding may introduce extra memory overhead; this means that more block RAMs are required.

Table 3 represents the use of logical units on FPGA. The utilization of block RAMS, Slices and DSPs are very related to the bank number. The average BRAMs improvement after using LTB is up to 21%. In the benchmarks DENOISE_2 and MOTION_C, the reduction of DSPs is up to 96% and 100%. In these cases, the partitioning number is reduced to a power of 2, which can be implemented as data shifting rather than using DSPs for the dividers. Although the use of the physical resources in DENOISE_1, DENOISE_2, BICUBIC_INTER, and SOBEL is reduced, the power estimation increases in these benchmarks, especially in SOBEL (about 48.85%). Based on our analysis of the transformed code, LTB uses more logic to implement the address generation for the array indices due to the extra padding size (It introduces an extra multiplication in each index). We could optimize it with some common address generation strategies (as the scheme proposed in [12]). However, in the benchmark SOBEL, as the flattening method uses 25% more block RAMs in this benchmark and the critical path is much longer than the one in LTB, the target CP (5ns) cannot be met.

In all, there is an average 21% reduction in BRAMs, 19% reduction in slices, 46% reduction in DSPs, and 14.69% more overhead in power. The CP has a small increase of 0.6% on average.

Table 1 Storage overhead of padding method

Array Size(# of data)	Padding Rate
<1000	0.0706
1000~5000	0.0281
5000~10000	0.0161
10000~20000	0.0116
>20000	0.0098

Table 2 High-level partitioning results

	Original II	Target II	Bank (Flatten)	Bank (LTB)	Padding size
DENOISE_1	5	1	6	5	64
DENOISE_2	8	1	10	8	128
MOTION_C	4	1	6	4	64
MOTION_LV	6	1	7	6	0
BICUBIC_INTER	4	1	6	5	64
SOBEL	9	1	12	9	64

Table 3 Synthesis experimental results

		Block RAM	Slice	DSP	CP (ns)	Power
DENOISE_1	Flatten	6	531	8	3.826	537
	LTB	5	441	8	4.451	685
	comp.(%)	**-16.7**	**-16.9**	**0**	**16.3**	**27.5**
DENOISE_2	Flatten	10	1114	75	4.995	1097
	LTB	8	767	3	4.563	1367
	comp.(%)	**-20**	**-31.1**	**-96**	**-8.6**	**24.6**
MOTION_C	Flatten	6	515	4	4.215	670
	LTB	4	255	0	4.068	484
	comp.(%)	**-33.3**	**-50.5**	**-100**	**-3.5**	**-27.8**
MOTION_ LV	Flatten	7	627	9	4.143	1263
	LTB	6	601	9	3.846	1026
	comp.(%)	**-14.3**	**-4.1**	**0**	**-7.2**	**-16.15**
BICUBIC_ INTER	Flatten	6	456	4	3.870	512
	LTB	5	441	4	4.451	672
	comp.(%)	**-16.7**	**-3.3**	**0**	**15**	**31.25**
SOBEL	Flatten	12	1302	105	5.222	1441
	LTB	9	1195	15	4.808	2145
	comp.(%)	**-25**	**-8.2**	**-85.7**	**-7.9**	**48.85**
AVERAGE(%)		**-21**	**-19**	**-46**	**0.6**	**14.69**

6. CONCLUSIONS

Memory partitioning is a crucial technology to enable data-level parallelism in FPGA designs. In this work we propose an automatic memory-partitioning method for multidimensional arrays. Linear transformation on the multidimensional array indices is introduced to extend the design space for the possible optimal solution. An optimal solution based on Ehrhart points counting and a heuristic solution based on memory padding are proposed. Experimental results demonstrate that compared with the state-of-art partitioning algorithm, our proposed algorithm can reduce the number of block RAMs by 21%.

7. ACKNOWLEDGMENTS

This work was supported in part by the National High Technology Research and Development Program of China 2012AA010902, RFDP 20110001110099 and 20110001120132, and NSFC 61103028. We would like to thank UCLA/PKU Joint Research Institute in Science and Engineering (JRI) and the support from Xilinx.

8. REFERENCES

[1] Bicubic interpolation http://www.mpi-hd.mpg.de/astrophysik/HEA/internal/Numerical_Recipes/f3-6.pdf

[2] Center for Domain-Specific Computing http://www.cdsc.ucla.edu/

[3] F.Balasa, H.Zhu, I.I.Lucian, "Computation of Storage Requirements for Multi-Dimensional Signal Processing Applications," Signal Processing Systemsm," in *IEEE Trans. Very Large Scale Integration Systems (TVLSI)*,VOL.15, No.4,2007.

[4] JM Software, H.264/AVC Software Coordination, http://iphome.hhi.de/suehring/tml/

[5] J. Cong, P. Zhang and Y. Zou, "Optimizing Memory Hierarchy Allocation with Loop Transformations for High-Level Synthesis", *Proceedings of the 49th Annual Design Automation Conference (DAC 2012)*, pp. 1233-1238, 2012.

[6] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization," in *ACM Trans. on Design Automation of Electronic Systems (TODAES),* 2011, Vol. 16 Issue 2, Article 15

[7] L. T. Yang,Y. Pan, et al, *High performance scientific and engineering computing: hardware/software support*, Springer, 2003

[8] M. Gupta, "Automatic Data Partitioning on Distributed Memory Multicomputers," 1992.

[9] P. Clauss, V. Loechner, "Parametric Parametric Analysis of Polyhedral Iteration Spaces," in *Journal of VLSI signal processing systems for signal, image and video technology*, Volume 19, Issue 2, pp 179-194, 1998.

[10] P. Feautrier, "Some efficient solutions for the affine scheduling problem, part I, one dimensional time," in *International Journal of Parallel Processing*, 21(6), December 1992

[11] P. Getreuer, "tvreg: Variational imaging methods for denoising, deconvolution, inpainting, and segmentation," online available: http://code.google.com/p/cdsc-image-processing-pipeline/downloads/list

[12] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory Partitioning and Scheduling Co-optimization in Behavioral Synthesis", in *Inter. Conf. on Computer-Aided Design (ICCAD),* 2012, pp. 488-495.

[13] Q. Liu, T. Todman, W. Luk, "Combining Optimizations in Automated Low Power Design," in *Proc.of Design, Automation and Test Europe(DATE),* 2010, pp. 1791-1796.

[14] ROSE compiler infrastructure, http://rosecompiler.org/

[15] S. Chatterjee, et al, "Generating Local Addresses and Communication Sets for Data-parallel Programs," *Journal of Parallel and Distributed Computing*,1995.

[16] S. Verdoolaege, H. Nikolov, and T. Stefanov, "pn: A Tool for Improved Derivation of Process Networks," *EURASIP Journal on Embedded Systems,* vol. 2007, pp. 1-13, 2007.

[17] Vivado High-Level Synthesis , http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/hls/index.htm

[18] Xilinx ISE Design Suite, http://www.xilinx.com/

[19] Y. Ben-Asher, N. Rotem, "Automatic Memory Partitioning: Increasing Memory Parallelism via Data Structure Partitioning," in *Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS),* 2010, pp, 155-162.

[20] Y. Wang, P. Zhang, X. Cheng, and J. Cong, "An Integrated and Automated Memory Optimization Flow for FPGA Behavioral Synthesis," in *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2012, pp. 257-262.

[21] Polylib, http://www.irisa.fr/polylib/

Appendix

1. The proof of Theorem 1

Assuming that there are two *d*-dimensional array references in the iteration domain as

$$R_0 = \begin{pmatrix} a_{0,0} & \cdots & a_{0,l} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \cdots & a_{d-1,l} \end{pmatrix} * (i_0, i_1, \dots, i_{l-1}, 1)^T \text{ and}$$

$$R_1 = \begin{pmatrix} b_{0,0} & \cdots & b_{0,l} \\ \vdots & \ddots & \vdots \\ b_{d-1,0} & \cdots & b_{d-1,l} \end{pmatrix} * (i_0, i_1, \dots, i_{l-1}, 1)^T.$$

The bank number mapping functions with a linear transformation vector $\vec{\alpha} = (\alpha_0, \alpha_1, \dots, \alpha_{d-1})$ are

$f(R_0) = (\vec{\alpha} * R_0) \% N$ and $f(R_1) = (\vec{\alpha} * R_1) \% N$

THEOREM 1. *Assuming that a d-dimensional array is accessed by two references* R_0 *and* R_1 *in an l-level loop nest, the array is cyclically partitioned into N banks with a linear transformation vector* $\vec{\alpha}$ *and a bank mapping function* f *so that the simultaneous accesses are not in conflict in the iteration domain, if*

$$gcd(\vec{\alpha} * \Delta_0{}^T, \vec{\alpha} * \Delta_1{}^T, \dots, \vec{\alpha} * \Delta_{l-1}{}^T, N) \nmid \vec{\alpha} * \Delta_l{}^T$$

where

$\Delta_k = (\Delta_{0,k}, \Delta_{1,k}, \dots \Delta_{d-1,k}), \Delta_l = (-\Delta_{0,l}, -\Delta_{1,l}, \dots -\Delta_{d-1,l}),$

$\Delta_{j,k} = a_{j,k} - b_{j,k}, \forall 0 \le k < l,\ 0 \le j < d$

Proof

The converse-negative proposition of theorem is proved as:

$$.\exists \vec{i} \text{ s.t. } f(R_0) = f(R_1)$$

$$\Leftrightarrow \vec{\alpha} * R_0 \equiv \vec{\alpha} * R_1 \bmod N$$

$$\Leftrightarrow \exists \vec{i}, k \text{ s.t.} \vec{\alpha} * \begin{pmatrix} \Delta_{0,0} & \cdots & \Delta a_{0,l-1} \\ \vdots & \ddots & \vdots \\ \Delta_{d-1,0} & \cdots & \Delta_{d-1,l-1} \end{pmatrix} * (i_0, i_1, \dots, i_{l-1})^T$$
$$+ kN = -\vec{\alpha} * (\Delta_{0,l}, \Delta_{1,l}, \dots \Delta_{d-1,l})^T$$

$$\Leftrightarrow \gcd(\vec{\alpha} * (\Delta_{0,0}, \Delta_{1,0}, \dots \Delta_{d-1,0})^T, \vec{\alpha} * (\Delta_{0,1}, \Delta_{1,1}, \dots \Delta_{d-1,1})^T,$$
$$\dots, \vec{\alpha} * (\Delta_{0,l-1}, \Delta_{1,l-1}, \dots \Delta_{d-1,l-1})^T, N) |$$
$$\vec{\alpha} * (\Delta_{0,l}, \Delta_{1,l}, \dots \Delta_{d-1,l})^T$$

$$\Leftrightarrow gcd(\vec{\alpha} * \Delta_0{}^T, \vec{\alpha} * \Delta_1{}^T, \dots, \vec{\alpha} * \Delta_{l-1}{}^T, N) | \vec{\alpha} * \Delta_l{}^T$$

where

$\Delta_k = (\Delta_{0,k}, \Delta_{1,k}, \dots \Delta_{d-1,k}), \Delta_l = (-\Delta_{0,l}, -\Delta_{1,l}, \dots -\Delta_{d-1,l}),$

$\Delta_{j,k} = a_{j,k} - b_{j,k}, \forall 0 \le k < l,\ 0 \le j < d$

2. Ehrhart's Points-Counting Theory

The following definitions and theorems are referenced from [9], as supplemental materials to section 4.2.1 to help understand the optimal approach.

Let *Q* denote the set of rational numbers and *Z* the set of integers. A convex polyhedron is defined by a finite set of linear inequalities:

$$P = \{\, x \epsilon Q^d \mid A \cdot x \le b \},$$

where A is a rational matrix and b a rational vector.

Definition 1 (homothetic-bordered system [9]). Let H_N, $N = (n_1, n_2, \dots, n_q)$, be a system defined by constraints of the form $\sum a_i x_i = \sum b_j n_j + c$, $\sum a_i x_i < \sum b_j n_j + c$, $\sum a_i x_i \le \sum b_j n_j + c$, where the a_i's, the b_i's and the c's are given integers, the x_i's are free variables and the n_i's are positive integral parameters.

Such a system is homothetic-bordered if and only if the polytope it defines has vertices whose coordinates are affine combinations of the parameters.

Counting the number of integer points is based on the decomposition of a parametric polytope into several homothetic-bordered systems, associated with validity domains.

Example

$$\begin{aligned} P_{n_1} &= \{x | x \ge 0, x \le n, 2x \le n+6\}, \\ P_{n_2} &= \{x | x \ge 0, n \ge 0, 2x \le n+6\}, \\ P_{n_3} &= \{x | 0 \le x \le n\} \end{aligned}$$

P_{n_2} and P_{n_3} are homothetic-bordered system and P_{n_1} is not homothethic-bordered system.

$$P_{n_1} = \cup \begin{cases} P_{n_2}, & \text{when } 0 \le n \le 6 \\ P_{n_3}, & \text{when } 6 \le n \end{cases}$$

Definition 2 (periodic number [9]). A one-dimensional periodic number $u(n) = [u_1, u_2, \dots, u_p]n$ is equal to the item whose rank is equal to *n* mod *p*, *p* is called the period of *u(n)*.

$$\mathrm{u(n)} = \begin{cases} u_1 \ ifn = 1(mod\ p), \\ u_2 \ ifn = 2(mod\ p), \\ \quad \dots \\ u_p \ ifn = 0(mod\ p). \end{cases}$$

Example

$$\begin{aligned} (-1)^n &= [-1, 1]_n \\ (-1)^{n-m} &= \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}_{(n,m)} \end{aligned}$$

Definition 3 (denominator [9]). The denominator of a rational point is the lowest common multiple of the denominators of its coordinates. The denominator of a rational polyhedron is the least common multiple of the denominators of its vertices.

Theorem 1 (Ehrhart's fundamental theorem [9]). *The enumerator* j_n *of any homothetic-bordered k polyhedron* P_n *is a polynomial in n of degree k if* P_n *is integral; and it is a pseudo-polynomial in n of degree k whose pseudo-period is the denominator of* P_n *if* P_n *is rational.*

EXAMPLE Bank mapping function: $f = (x_0 + 2x_1)\%5$, $0 \le x_0, x_1 \le 64$, for (x_0, x_1) = (32,15), find the inner bank address.

There are two polytopes: base polytope and offset polytope

- The base polytope is

$$\begin{cases} x_0 + 2x_1 + 5k = \mathrm{c} \\ 0 \le x_0, x_1 \le 64 \\ k \ge 1 \\ c \ge 0 \end{cases}$$

There are four Ehrhart polynomials for the base polytope. For different domain of d, they are:

Domain1: c -197 >= 0

Ehrhart Polynomial: $L_{base}(c)$ =845

Domain2: c -133 >= 0 and - c + 197 >= 0

Ehrhart Polynomial:

$$L_{base}(c) = -\frac{1}{20} \times c^2 + \frac{98}{5} \times c + \left[-1076, -\frac{21511}{20}, -1076, -\frac{21507}{20}, -\frac{5378}{5}, -\frac{4303}{4}, -\frac{5379}{5}, -\frac{4303}{4}, -\frac{5378}{5}, -\frac{21507}{20}\right]$$

Domain3: c -69 >= 0 and - c + 133 >= 0

Ehrhart Polynomial: $L_{base}(c) = \frac{13}{2} \times c + [-218, -\frac{435}{2}]$

Domain4: - c + 69 >= 0 and c -5 >= 0

Ehrhart Polynomial:

$$L_{base}(c) = \frac{1}{20} \times c^2 + \left(-\frac{1}{10} \times c\right) + \left[0, \frac{1}{20}, 0, -\frac{3}{20}, -\frac{2}{5}, \frac{1}{4}, -\frac{1}{5}, \frac{1}{4}, -\frac{2}{5}, -\frac{3}{20}\right]$$

- The offset polytope is

$$\begin{cases} x_0 + 2x_1 = x_0' + 2x_1' \\ 0 \le x_0 \le x_0' \\ 0 \le x_0', \ x_1' \le 64 \end{cases}$$

$$L_{offset}(x_0') = \frac{1}{2} \times x_0' + [1, \frac{1}{2}]$$

(x_0, x_1) = (32,15), c=62,

g(32,15) = $L_{base}(62) + L_{offset}$=186 + 17 =203

3. Detailed descriptions of the benchmarks

The detailed description of the benchmarks is listed in Table 2. DENOISE_1 and DENOISE_2 are from the Rician-denoise algorithm [11] from medical image applications, and their access patterns are shown in Fig. 5(a), Fig. 5(b). DENOISE_1 and DENOISE_2 are the original access patterns in the application. DENOISE_2 is the access pattern by unrolling DENOISE_1 by 2. MOTION_LV and MOTION_C are the different loop kernels of motion compensation from official H.264 decoder JM 14.0 [4]. MOTION_C is the interpolation for the chroma components. MOTION_LV is the motion compensation for the luma samples in the video frame in the vertical direction. Their access patterns are shown in Fig. 5(c) and Fig. 5(d). BICUBIC_INTER [1] is from bicubic interpolation process. And SOBEL [16] is from Sobel edge detection algorithm. The access patterns of them are illustrated in Fig. 5(e) and Fig. 5(f).

Table 4 Benchmark Description

	Benchmark description
DENOISE_1	2D Rician-denoise, as Fig. 5(a)
DENOISE_2	2D Rician-denoise, with loop titling, as Fig. 5(b)
MOTION_C	H.264 motion compensation for chroma samples, as Fig. 5(c)
MOTION_LV	H.264 Motion compensation for luma samples in horizontal direction, as Fig. 5(d)
BICUBIC_INTER	Bicubic interpolation, as Fig. 5(e)
SOBEL	2D Sobel edge detection algorithm, as Fig. 5(f)

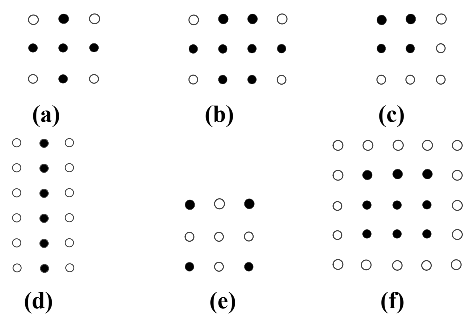


Fig. 5 The access patterns of the benchmarks: (a) DENOISE_1 (b) DENOISE_2 (c) MOTION_C (d) MOTION_LV (e) BICUBIC_INTER (f) SOBEL