Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis

Yuxin Wang[1]
ayerwang@pku.edu.cn

Peng Li[1]
peng.li@pku.edu.cn

Jason Cong [2,3,1,*]
cong@cs.ucla.edu

[1]Center for Energy-Efficient Computing and Applications, Peking University, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]PKU/UCLA Joint Research Institute in Science and Engineering

ABSTRACT

The significant development of high-level synthesis tools has greatly facilitated FPGAs as general computing platforms. During the parallelism optimization for the data path, memory becomes a crucial bottleneck that impedes performance enhancement. Simultaneous data access is highly restricted by the data mapping strategy and memory port constraint. Memory partitioning can efficiently map data elements in the same logical array onto multiple physical banks so that the accesses to the array are parallelized. Previous methods for memory partitioning mainly focused on cyclic partitioning for single-port memory. In this work we propose a generalized memory-partitioning framework to provide high data throughput of on-chip memories. We generalize cyclic partitioning into block-cyclic partitioning for a larger design space exploration. We build the conflict detection algorithm on polytope emptiness testing, and use integer points counting in polytopes for intra-bank offset generation. Memory partitioning for multi-port memory is supported in this framework. Experimental results demonstrate that compared to the state-of-art partitioning algorithm, our proposed algorithm can reduce the number of block RAM by 19.58%, slice by 20.26% and DSP by 50%.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design

Keywords

high-level synthesis, memory partitioning, polyhedral model

*In addition to being a faculty member at UCLA, Jason Cong is also a co-director of the PKU/UCLA Joint Research Institute and a visiting professor of Peking University.

FPGA'14, February 26–28, 2014, Monterey, CA, USA.

http://dx.doi.org/10.1145/2554688.2554780 .

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. In addition to their traditional use, FPGA devices are increasingly used as hardware accelerators to speed up the performance of energy-critical applications in heterogeneous computing platforms. A major obstacle that FPGA accelerators must overcome is that of finding an efficient programming model. Traditional register-transfer level (RTL) programming requires expertise with the hardware description language (e.g., VHDL, Verilog), and more importantly, with the hardware mindset, including, the underlying concurrent hardware programming model and the trade-offs between performance and resource utilization. Manual RTL design is time-consuming, error-prone and difficult to debug. Implementing a simple algorithm often takes weeks or months, even for an expert.

High-level synthesis (HLS) is a key technology for breaking the programming wall of FPGA-based accelerators. By transforming algorithms written in high-level languages (e.g., C, C++, SystemC), HLS can significantly shorten the learning curve and improve programming productivity. After over 30 years of joint effort by academia and industry, HLS promises to be a critical bridging technology that offers high productivity and quality of results for both the semiconductor and FPGA industries. A number of state-of-art commercial tools have been developed and gradually accepted by the traditional hardware designers, such as Vivado_HLS from Xilinx [9] (based on AutoPilot [13, 23]), C-to-Silicon from Cadence [2], Catapult C from Calypto [3], Synphony from Synopsys [8], and Cynthesizer [4] from Forte.

Although off-the-shelf commercial HLS tools are capable of generating high-quality circuits, there is still a significant performance gap compared to manually customized RTL designs, even for programs with regular affine loop bounds and array accesses. One important reason for this performance gap is the redundant off-chip and on-chip memory accesses and inefficient loop pipelining with the presence of loop dependence and resource restraint. Currently, most of the tools support versatile efficient parallelization flows on a data path, such as loop pipelining and loop unrolling. FPGAs are capable of providing enough computational units for high-computation parallelism and plenty of on-chip memory resources for parallelized data accesses, while the simultaneous data access is highly restricted by the data mapping

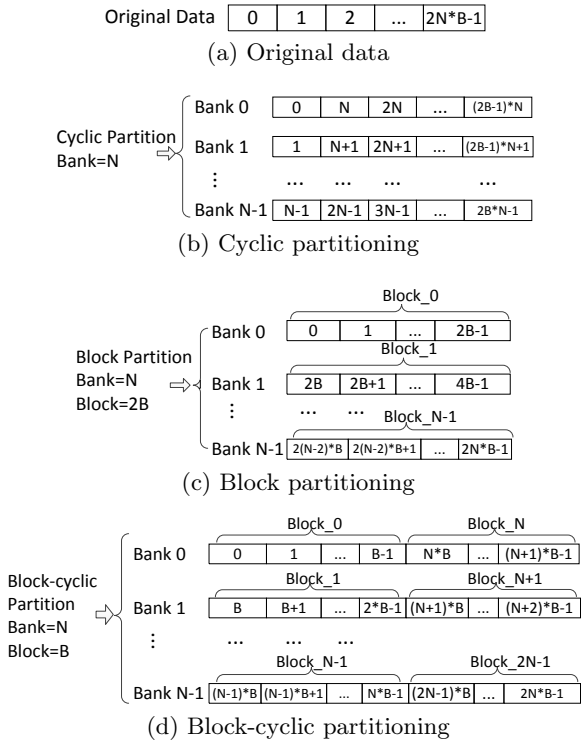


Figure 1: Memory partitioning schemes: (a) original data (b) cyclic partitioning (c) block partitioning (d) block-cyclic partitioning

strategy and memory port constraint. Typical block RAMs (BRAMs) in FPGAs have limited ports to feed the highly parallelized execution units. Memory becomes a crucial bottleneck that impedes performance enhancement when multiple data elements from the same array are required simultaneously. How to supply the computational units with the required high-speed data streams is a major challenge.

It is unrealistic to increase the number of ports of block RAMs on FPGAs. Even in an ASIC design, increasing the ports' number induces quadratic growth in complexity and area. Duplicating the data elements into multiple copies can support simultaneous data accesses [18], but it may have significant area and power overhead and introduce memory consistency problems.

A comparatively 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. This method provides the same effects as memory duplication, largely saving the storage requirement and having no problem with multiple data copies. According to the classification in [10], the regular memory partitioning methods are classified as cyclic partitioning, block partitioning, and block-cyclic partitioning. We depict them in Fig. 1(b), Fig. 1(c), and Fig. 1(d) respectively, where each square denotes a data element in the array.

Memory partitioning in high-level synthesis has been studied in several related works. A scheduling-based automated flow is proposed in [12] to support multiple simultaneous affine memory accesses. The algorithm can be extended to efficiently support memory references with modulo operations with limited memory paddings [22]. The work in [17] schedules memory accesses in different loop iterations to find the optimal or near-optimal partitioning in a larger design exploration space. These memory-partitioning methodologies are able to find the optimal partitioning for one-dimensional arrays under different circumstances. However, many designs for FPGAs are specified by nested loops with multidimensional arrays, such as image and scientific computing. The work in [21] provides a linear transformation-based (LTB) method for multidimensional memory partitioning. However, the method is limited to cyclc partitioning. The access conflict analysis algorithm eliminates the information of iteration vectors in the index and only provides a sufficient but unnecessary condition, which may lead to suboptimality in some cases.

In this work we provide a generalized memory partitioning (GMP) method. The main contributions of this work are described as follows:

1. We formulate the memory partitioning using a polyhedral model, including various memory-partitioning schemes, and supporting partitioning for multi-port memories.

2. We transform the problem of detecting the access conflict between a pair of references to an equivalent problem of polytope emptiness testing. We also formulate the intra-bank offset generation as a problem of counting the integer points in the polytopes in lexicographic order.

To our knowledge, we are the first to use a polyhedral model to solve the bank access conflict problem in loop nest. We are also the first to use the polytope emptiness test for access conflict detection. In addition, we propose a resource estimation model, which enables the choice of an optimal partitioning automatically through design space exploration.

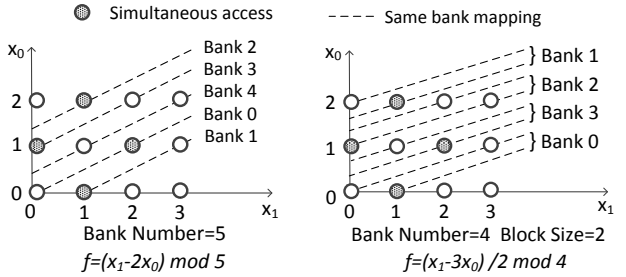
The remainder of this paper is organized as follows: Section 2 gives the motivational examples for our work; Section 3 formulates the problem; Section 4 introduces the bank-mapping algorithm; and Section 5 describes the theory for intra-bank offset generation. Section 6 presents the design space exploration; the experimental results is discussed in Section 7; this is followed by conclusions in Section 8.

2. MOTIVATIONAL EXAMPLES

In this section we use a motivational example to explain the limitations of previous work [21]. Our motivational example stems from a critical loop kernel of a 2D denoise algorithm derived from medical image processing [14]. As shown in Fig. 2(a), there are four accesses to a two-dimensional array in the loop nest. To improve the processing throughput of the loop kernel, we pipeline the execution of successive inner-loop iterations, which means that multiple accesses to the same array will happen in one clock cycle. We assume that only one array element can be read from a physical memory in each clock cycle (the case with multi-port memories is considered in Section 4.3). If all the array elements are stored in one physical memory, fetching all the required data elements in an iteration requires four clock cycles. Thus, the performance will be highly impacted. Memory partitioning allocates the array elements into multiple banks to reduce the access conflict.

```
int A[w0][w1];
for (j=1; j<w0-1; j++)
 for (i=1; i<w1-1; i++)
  foo(A[j][i-1], A[j-1][i ], A[j+1][i], A[j][i+1]);
```

(a) Loop kernel



(b) Cyclic partitioning (c) Block-cyclic partitioning

Figure 2: Denoise: (a) loop kernel, (b) cyclic partitioning, (c) block-cyclic partitioning

The memory-partitioning method in a previous work [21] can solve this problem by applying linear transformation-based cyclic partitioning on the multidimensional array. Using this method, we can always achieve conflict-free access with five memory banks. Fig. 2(b) illustrates the partitioning with five banks, where x_0 and x_1 are the two dimensions of array A[1]. The points represent the array elements, and the shaded points are the data elements accessed in the same loop iteration. The data elements on the same dotted line are mapped to the same memory bank. The partitioning function is $f = (x_1 - 2x_0)\ mod\ 5$. As we see in the figure, four simultaneous accesses are in different banks, so there is no conflict.

However in this case, the use of cyclic partitioning alone will not achieve the ideal optimal result. The ideal minimum bank number is four for Fig. 2(a), as there are only four accesses in the inner loop. Note that previous HLS memory-partitioning work [12,17,22] also use only cyclic partitioning, and thus suffer the same limitations. Block-cyclic provides a larger design exploration space and can solve this problem, as shown in Fig. 2(c). The ideal conflict-free partitioning with four banks can be achieved by using a partition block size of two.

Moreover, the current block RAMs on the real FPGA platforms always have dual ports. Previous work [12] proposes a scheduling-based method for partitioning for multi-port memory, but the mapping relationship between the ports and references is fixed. This may lead to a suboptimal partitioning result. Our generalized partitioning model based on polyhedral model can be easily extended for multi-port memory.

3. PROBLEM FORMULATION

In this section we present the definitions and problems of memory partitioning. The important variables in the following descriptions are listed in Table. 1. We introduce a polyhedral model to define the problem. The polyhedral model is a powerful mathematical framework based on parametric linear algebra and integer linear programming. It provides a flexible representation for expressing the loop nests.

[1]Note that the order of the dimensions is different from that in [21].

Table 1: Symbol table

Variables	Meaning
N	Partition factor, representing the number of logic banks used after memory partitioning
B	Partition block size
P	Memory port number
l	Level of loop nest
d	Number of dimensions of the array
m	Number of array references in the inner loop
$\mathcal{D}$	Iteration domain
$\mathcal{M}$	Data domain
$\vec{i}$	Iteration vector
$\vec{x}$	Array index vector
$\vec{\alpha}$	Partition vector
q	Padding size
i, j, k, t	Temporal variables
$\mathbb{Z}$	Integer set
w_k	The k-th dimensional size of the array

Definition 1. (Polyhedron) A set $P \in Q^d$ is a polyhedron if there exists a system of a finite number of inequalities $A \cdot \vec{x} \leq b$ such that

$$P = \{\vec{x} \in Q^d | A \cdot \vec{x} \leq \vec{b}\},$$

where Q denotes the set of rational numbers, $\vec{x}$ is a d-dimensional vector in d-dimensional space Q^d. A is a rational matrix and $\vec{b}$ is a rational vector. A polytope is a bounded polyhedron. We define the iteration domain, data domain and affine array references as polytopes.

Definition 2. (Iteration Domain [15]) Given an l-level loop nest, the iteration domain $\mathcal{D}$ is formed by the iteration $\vec{i} = (i_0, i_1, ..., i_{l-1})^T$ within the loop bounds.

Definition 3. (Data Domain) Given a d-dimensional array, the data domain $\mathcal{M}$ is bounded by the array size, where $\forall 0 \leq k < d$, the k-th dimensional size is w_k.

Definition 4. (Affine Memory Reference) A d-dimensional affine memory reference $\vec{x} = (x_0, x_1, ..., x_{d-1})^T$ is represented as the following linear combinations

$$\vec{x} = A_{d\times l} \cdot \vec{i} + C,$$

$$A_{d\times l} = \begin{pmatrix} a_{0,0} & \cdots & a_{0,l-1} \\ \vdots & \ddots & \vdots \\ a_{d-1,0} & \cdots & a_{d-1,l-1} \end{pmatrix}, C = \begin{pmatrix} a_{0,l} \\ \vdots \\ a_{d-1,l} \end{pmatrix}$$

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

Example 1. An affine array reference $A[i_0][i_1 + 1]$ is represented as $\vec{x} = (i_0, i_1 + 1)^T$, where

$$\vec{x} = \begin{pmatrix} i_0 \\ i_1 + 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i_0 \\ i_1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

With these definitions, we can formulate the bank mapping of the array elements in the polyhedral model. Our memory partitioning consists of two mapping problems: bank mapping and intra-bank offset mapping.

Definition 5. (Memory Partitioning) A memory partitioning of an array is described as a pair of mapping functions $(f(\vec{x}), g(\vec{x}))$, where $f(\vec{x})$ assigns a bank for the data element, and $g(\vec{x})$ generates the corresponding intra-bank offset.

A bank access conflict between two references $\vec{x}_j$ and $\vec{x}_k$ $(0 \leq j < k < m)$ is represented as $\exists \vec{x} \in \mathcal{D}$, s.t.

$$f(\vec{x}_j) = f(\vec{x}_k).$$

This means the references intend to access the same bank in the same clock cycle. We use Problem 1 to formulate the bank mapping problem (for single-port memories).

PROBLEM 1. *(Bank Minimization) Given a l-level loop in the iteration domain $\mathcal{D}$ with m affine memory references $\vec{x}_0, ..., \vec{x}_{m-1}$ on the same array, find the partition factor N such that:*

$$Minimize : N = max_{\leq i<m}\{f(\vec{x}_i)\} \quad (1)$$

$$\exists \vec{i} \in \mathcal{D}, 0 \leq j < k < m, f(\vec{x}_j) \neq f(\vec{x}_k). \quad (2)$$

Eqn. (1) defines the objective function of memory partitioning, and Eqn. (2) ensures no access conflict between any two references. After bank mapping, a data element in the original array should be allocated a new intra-bank location. For correctness, two different array elements will be either mapped onto different banks or the same bank with different intra-bank offsets. An intra-bank offset function is valid if and only if

$$\forall \vec{x}_j, \vec{x}_k \in \mathcal{M}, \vec{x}_j \neq \vec{x}_k \Leftrightarrow (f(\vec{x_j}), g(\vec{x_j})) \neq (f(\vec{x_k}), g(\vec{x_k})),$$

which means either

$$f(\vec{x}_j) \neq f(\vec{x}_k), or\ f(\vec{x}_j) = f(\vec{x}_k),\ g(\vec{x}_j) \neq g(\vec{x}_k).$$

Considering storage requirement, we also want to minimize the largest offset of each bank. The intra-bank offset mapping problem is formulated as Problem 2.

PROBLEM 2. *(Storage Minimization) Given an l-level loop in the iteration domain with m affine memory references $\vec{x}_0, ..., \vec{x}_{m-1}$ on the same array and a partition factor N, find an intra-bank offset mapping function g with minimum storage requirement $\mathcal{S}$ such that:*

$$Minimize : \sum_{j=0}^{N-1} max_{\leq i<m, f(\vec{x}_i)=j}\{g(\vec{x}_i)\} \quad (3)$$

$$\forall \vec{x}_j, \vec{x}_k \in \mathcal{M}, (f(\vec{x_j}), g(\vec{x_j})), \neq (f(\vec{x_k}), g(\vec{x_k})). \quad (4)$$

Eqn. (3) defines the objective function of partitioning with minimum storage overhead, and Eqn. (4) is responsible for the valid partition detection.

4. BANK MAPPING

In this section we describe how we automatically map a array in a loop nest into separate memory banks to enable parallelized memory access. We use a loop initiation interval (II) of loop pipelining to measure the throughput. It represents the clock cycles between the successive iterations and reflects the parallelism of on-chip memory access. For a fully pipelined loop with all accesses parallelized, II is one. This is the performance target in this paper. For other constant loop initiation intervals, we use conflict analysis together with scheduling to find a proper partitioning (as presented in [12]).

In our GMP method, we consider the block-cyclic partitioning scheme, for it covers all three schemes mentioned in our introduction. The bank mapping function for memory reference $\vec{x}$, with a partition vector $\vec{\alpha} = (\alpha_0, \alpha_1, ..., \alpha_{l-1})$, $\alpha_i \in \mathbb{Z}$, is described as Eqn. (5).

$$f(\vec{x}) = \left\lfloor \frac{\vec{\alpha} \cdot \vec{x}}{B} \right\rfloor \ mod\ N \quad (5)$$

N is the partition factor representing the total bank number, and B is the partition block size. The bank mapping function for cyclic partitioning is $f(\vec{x}) = (\vec{\alpha} \cdot \vec{x})\ mod\ N$. From a geometrical point of view, $\vec{\alpha} \cdot \vec{x}$ represents a sequence of hyperplanes in the data domain. The mapping function $f(\vec{x})$ assigns the points on the hyperplanes to different memory banks.

Given a partition factor N, a block size B, and a partition vector $\vec{\alpha}$, the conflict detection process is executed between each pair of references. We propose a method using polytope emptiness testing. For simplicity, we first discuss the conflict detection in cyclic partitioning under a single-port memory constraint. We will introduce the algorithm for block-cyclic partitioning and partitioning for multi-port memory as extensions to the framework.

4.1 Generalized Conflict Detection

Considering the iteration domain, integer linear programming is optimal in access conflict analysis. In this section we use a polyhedral model for access conflict detection in the iteration domain with given parameters, including partition factor, block and vector. The framework is general for extending to multiple loop kernels. We introduce the concept of a conflict polytope expressing the access conflict necessarily occuring between a pair of references in any iteration in the iteration domain. We assume that two d-dimensional affine memory references are $\vec{x}_0 = A_0 \cdot \vec{i} + C_0$ and $\vec{x}_1 = A_1 \cdot \vec{i} + C_1$.

Definition 6. (Conflict Polytope) A conflict polytope of two simultaneous references $\vec{x}_0$ and $\vec{x}_1$ is a parametric polytope restricted to the iteration domain $\mathcal{D}$ as

$$\mathcal{P}_{conf}(\vec{x}_0, \vec{x}_1) = \{\vec{i} | \forall \vec{i} \in \mathcal{D}, f(\vec{x}_0) = f(\vec{x}_1)\}.$$

Obviously, if $\forall \vec{i} \in \mathcal{D}$, $f(\vec{x}_0) \neq f(\vec{x}_1)$, $\mathcal{P}_{conf}(\vec{x}_0, \vec{x}_1)$ is empty.

THEOREM 1. *Given two memory references $\vec{x}_0$ and $\vec{x}_1$ in the same inner-loop iteration, they are conflict free in any iteration if and only if their conflict polytope $P_{conf}(\vec{x}_0, \vec{x}_1)$ is empty.*

PROOF. Proof omitted due to the page limit. □

With the bank mapping function $f(\vec{x}) = (\vec{\alpha} \cdot \vec{x})\ mod\ N$, two distinct data elements $\vec{x}_0$ and $\vec{x}_1$ are mapped to the same bank if and only if

$$f(\vec{x}_0) = f(\vec{x}_1) \Leftrightarrow \vec{\alpha} \cdot \vec{x}_0\ mod\ N = \vec{\alpha} \cdot \vec{x}_1\ mod\ N$$
$$\Leftrightarrow \exists k \in \mathbb{Z}, s.t.\ \vec{\alpha} \cdot \vec{x}_0 + Nk = \vec{\alpha} \cdot \vec{x}_1,$$

where $k \in \mathbb{Z}$ is a variable. The conflict polytope for cyclic partitioning is formulated as

$$\mathcal{P}_{conf}: \begin{cases} \vec{\alpha}\cdot(A_0 - A_1)\cdot\vec{i} + \vec{\alpha}\cdot(C_0 - C_1) + Nk = 0 \\ \vec{i} \in \mathcal{D} \\ k \in \mathbb{Z} \end{cases}$$

where $\vec{i} = (i_0, i_1, ..., i_{l-1})^T$ and k are variables. Example 2 is a detailed example. We apply an emptiness test on the polytope [19] to detect the access conflict.

Example 2. For the references $A[i_0+1][i_1]$ and $A[i_0][i_1+1]$ with a partition vector $\vec{\alpha} = (2, 1)$ and partition factor $N = 2$, and the iteration domain $0 \le i_0, i_1 \le 63$, $\exists k \in \mathbb{Z}$

$$\begin{aligned} f(\vec{x}_0) = f(\vec{x}_1) \Leftrightarrow & 2(i_0+1) + i_1 + 2k = 2i_0 + i_1 + 1 \\ \Leftrightarrow & 2k + 1 = 0 \end{aligned}$$

Thus the conflict polytope is formed by $2k + 1 = 0$ and the iteration domain as

$$\mathcal{P}_{conf}: \left(\begin{array}{cccc} 0 & 0 & 2 & 1 \\ \hline 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 63 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 63 \end{array}\right) \cdot \begin{pmatrix} i_0 \\ i_1 \\ k \\ 1 \end{pmatrix} \begin{array}{l} = 0 \\ \hline \ge \vec{0} \end{array}$$

The polytope is empty. According to Theorem 1, no conflict exists between these two array references. For multiple references in the inner-loop, the conflict detection is executed between each pair of the references. This means for m references, C_m^2 conflict polytopes are constructed for non-conflict access among all the references. Because of the exponential complexity of the polyhedral emptiness tests, this pair-wise conflict test, which keeps a moderate numbers of dimensions at each run, is a more scalable approach.

4.2 Extension for Block-Cyclic Partitioning

In block-cyclic partitioning, a reference $\vec{x}$ is first linearized as $\vec{\alpha}\cdot\vec{x}$, and partitioned into blocks by $\lfloor\frac{\vec{\alpha}\cdot\vec{x}}{B}\rfloor$. Then the blocks are cyclically allocated to banks. The access conflict between $\vec{x}_0$ and $\vec{x}_1$ is formed as

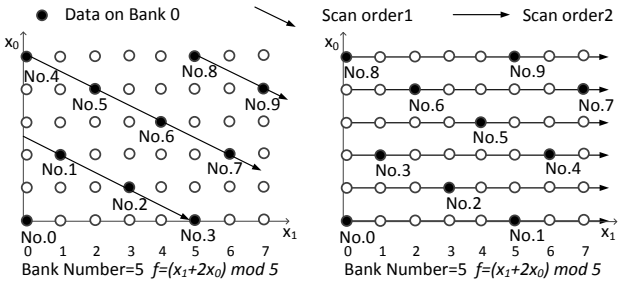
$$\begin{aligned} f(\vec{x}_0) = f(\vec{x}_1) \Leftrightarrow & \lfloor\frac{\vec{\alpha}\cdot\vec{x}_0}{B}\rfloor \ mod\ N = \lfloor\frac{\vec{\alpha}\cdot\vec{x}_1}{B}\rfloor \ mod\ N \\ \Leftrightarrow & \exists k_0 \in \mathbb{Z}, s.t.\ \lfloor\frac{\vec{\alpha}\cdot\vec{x}_0}{B}\rfloor + Nk_0 = \lfloor\frac{\vec{\alpha}\cdot\vec{x}_1}{B}\rfloor. \end{aligned}$$

As this formulation is not linear, it cannot be represented by a polyhedral model. Further linearizing is required to express the partition blocks as $\exists k_0, k_1 \in \mathbb{Z}, s.t.$

$$\begin{cases} BNk_0 + Bk_1 \le \vec{\alpha}\cdot\vec{x}_0 \le BNk_0 + B(k_1+1) - 1 \\ Bk_1 \le \vec{\alpha}\cdot\vec{x}_1 \le B(k_1+1) - 1 \end{cases}$$

With the above inequalities expressing the linearized space inside the blocks and among the banks, the conflict polytope for block-cyclic partitioning is formulated as below, where $\vec{i} = (i_0, i_1, ..., i_{l-1})^T, k_0$ and k_1 are variables.

$$\mathcal{P}_{conf}: \begin{cases} -\vec{\alpha}\cdot A_0\cdot\vec{i} - \vec{\alpha}\cdot C_0 + B(Nk_0 + k_1 + 1) - 1 \ge 0 \\ -\vec{\alpha}\cdot A_1\cdot\vec{i} - \vec{\alpha}\cdot C_1 + B(k_1+1) - 1 \ge 0 \\ \vec{\alpha}\cdot A_0\cdot\vec{i} + \vec{\alpha}\cdot C_0 - B(Nk_0 + k_1) \ge 0 \\ \vec{\alpha}\cdot A_1\cdot\vec{i} + \vec{\alpha}\cdot C_1 - Bk_1 \ge 0 \\ \vec{i} \in \mathcal{D} \\ k_0, k_1 \in \mathbb{Z} \end{cases}$$



(a) Order 1 (b) Order 2

Figure 3: Examples of data ordering

Example 3 compares cyclic partitioning and block-cyclic partitioning by giving different block size.

Example 3. For two references $A[i_0][i_1 - 1]$ and $A[i_0][i_1 + 1]$, with a partition vector $\vec{\alpha} = (2, 1)$, partition factor $N = 2$, the partition block size $B = 1$, and the iteration domain $0 \le i_0, i_1 \le 63$, the conflict polytope is non-empty, which means conflict exists between two references when using two banks. But with a partition vector $\vec{\alpha} = (2, 1)$, partition factor $N = 2$, the partition block size $B = 2$, the conflict polytope is empty. According to Theorem 1, no conflict exists between two references.

4.3 Extension for Multi-Port Memories

The conflict polytope built for multi-port partitioning follows the same principle. Take cyclic partitioning under the port constraint $P = 2$, for example, assuming three simultaneous accesses $\vec{x}_0$, $\vec{x}_1$ and $\vec{x}_2$, where $\vec{x}_2 = A_2\cdot\vec{i} + C_2$. The access conflict is formed by

$$f(\vec{x}_0) = f(\vec{x}_1)\ and\ f(\vec{x}_1) = f(\vec{x}_2).$$

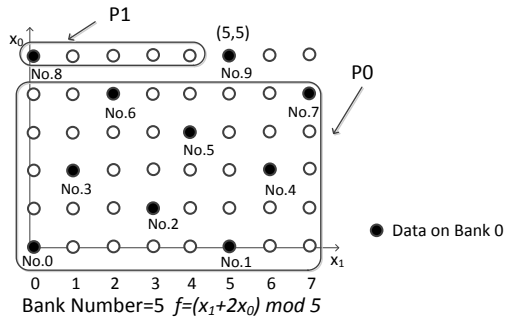
The conflict polytope is constructed as

$$\mathcal{P}_{conf}: \begin{cases} \vec{\alpha}\cdot(A_0 - A_1)\cdot\vec{i} + \vec{\alpha}\cdot(C_0 - C_1) + Nk_0 = 0 \\ \vec{\alpha}\cdot(A_1 - A_2)\cdot\vec{i} + \vec{\alpha}\cdot(C_1 - C_2) + Nk_1 = 0 \\ \vec{i} \in \mathcal{D} \\ k_0, k_1 \in \mathbb{Z} \end{cases}$$

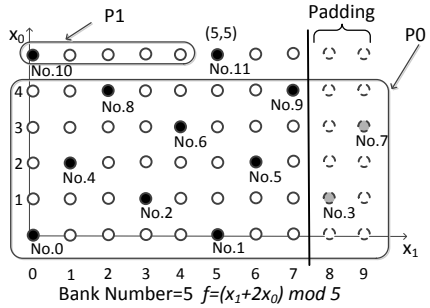
where $\vec{i} = (i_0, i_1, ..., i_{l-1})^T, k_0$ and k_1 are variables. Given m references and P memory ports, we need to detect the conflict among every $P + 1$ references by building C_m^{P+1} polytopes, and testing the emptiness of each of them.

5. INTRA-BANK OFFSET GENERATION

The function $g(\vec{x})$ maps a multidimensional array address to a non-negative integer as the corresponding intra-bank offset after bank mapping, following the partitioning validity principle. An optimal approach without increasing the original size is to scan and order the data elements on the same bank in a certain sequence. Fig. 3 shows an example of data ordering, where the black points are data elements assigned to Bank 0. Fig. 3(a) and Fig. 3(b) shows different scanning order. The corresponding intra-bank offset mapping functions are different as well.



(a) Optimal approach



(b) Heuristic approach

Figure 4: An example of generating intra-bank offset: (a) optimal approach (b) heuristic approach

In this section we propose a method based on polytope integer points counting to generate the intra-bank offset mapping function. First, we introduce the concept of bank polytope for the references mapped to the same bank.

Definition 7. (Bank Polytope [16]) Given a d-dimensional array reference $\vec{x}$, $\mathcal{P}_{bank}(\vec{x})$ is a bank polytope of $\vec{x}$ in the data domain $\mathcal{M}$ defined as

$$\mathcal{P}_{bank}(\vec{x}) = \{\vec{y} | \forall \vec{y} \in \mathcal{M}, f(\vec{x}) = f(\vec{y})\}.$$

With a partition factor N, there are N bank polytopes in total. The minimum total storage requirement for bank $f(\vec{x})$ is $C(\mathcal{P}_{bank}(\vec{x}))$, where $C(\mathcal{P})$ is a function calculating the number of integer points in the polytope $\mathcal{P}$.

Definition 8. (Lexicographic Order) A lexicographic order $\prec_{lex}$ on a d-dimensional set $\mathcal{M}$ is a relation, where for $\forall \vec{x}, \vec{y} \in \mathcal{M}$, $\vec{x} = (x_0, x_1, ..., x_{d-1})$ and $\vec{y} = (y_0, y_1, ..., y_{d-1})$,

$$\vec{y} \prec_{lex} \vec{x}$$

$$\Leftrightarrow \exists 1 < t < d, \forall 0 \leq i < t, (x_i = y_i) \wedge (y_t < x_t).$$

We can define a valid and optimal intra-bank offset mapping function with minimum storage requirement by using the lexicographic ordering number as the intra-bank offset as

$$g(\vec{x}) = C(\{\vec{y} | \vec{y} \in \mathcal{P}_{bank}(\vec{x}), \vec{y} \prec_{lex} \vec{x}\}).$$

5.1 Integer Points Counting in Polytopes

We illustrate a two-dimensional example in Fig. 4(a). The intra-bank offset for $(5,5)$ is generated by counting the integer points in the bank polytope $\mathcal{P}_{bank}(5,5)$ in lexicographic order. The data elements on Bank 0 are represented as the dark points. The polytope $\mathcal{P}_{bank}(5,5)$ is dvided into two polytopes, $\mathcal{P}_0$ and $\mathcal{P}_1$. There are eight integer points in $\mathcal{P}_0$, and one point in $\mathcal{P}_1$. As a result, the intra-bank offset for $(5,5)$ is nine. We define $\mathcal{P}_0$ and $\mathcal{P}_1$ together as an ordered polytopes set. Given a d-dimensional address $\vec{x} = (x_0, x_1, ..., x_{d-1})^T$ in the data domain, its ordered polytopes set $\mathcal{S}_{P_t(\vec{x})}$ is constructed by a union of polytopes as

$$\mathcal{S}_{P_t(\vec{x})} = \{\mathcal{P}_t(\vec{x}) \mid \forall 1 \leq t \leq d\}, where$$

$$\mathcal{P}_t(\vec{x}) = \{\vec{y} | \vec{y} \in \mathcal{P}_{bank}(\vec{x}), \forall 0 \leq i < t, (x_i = y_i) \wedge (y_t \prec_{lex} x_t)\}.$$

$\mathcal{P}_t(\vec{x})$ is a sub-polytope of $\mathcal{P}_{bank}(\vec{x})$. $\mathcal{S}_{P_t(\vec{x})}$ is organized by generating $\mathcal{P}_t(\vec{x})$ in lexicographic order.

THEOREM 2. *Given an ordered polytopes set $\mathcal{S}_{P_t(\vec{x})}$ of $\vec{x}$, a valid and optimal intra-bank offset generation function for $\vec{x}$ with minimum storage requirement is $\forall 0 \leq i < t, \mathcal{P}_t(\vec{x}) \in \mathcal{S}_{P_t(\vec{x})}$,*

$$g(\vec{x}) = \sum C(\mathcal{P}_t(\vec{x}))$$

PROOF. Proof omitted due to the page limit. □

Thus, we can convert the intra-bank offset generation problem to an equivalent problem on counting the total number of elements in several parameterized polytopes. Counting integer points in polytopes is a fundamental mathematical problem, and has a wide application in analysis and transformations of nested loop programs. Several automatic algorithms and libraries have been previously provided, such as Ehrhart's theorem [11] and the Barvinok library [20]. In this paper we use Ehrhart's method presented by [11] for computing the parametric vertices. It generates the Ehrhart polynomial as a parametric expression of the number of integer points. Example 4 presents a detailed example for generating the intra-bank offset by using this method [11].

Example 4. Given the array reference vector $\vec{x} = (x_0, x_1)$, the data domain $0 \leq x_0, x_1 \leq 7$, a partition vector $\vec{\alpha} = (2,1)$, a partition factor $N = 5$, and partition block size $B = 1$, the polytope $\mathcal{P}_0$ is formulated by

$$\begin{cases} 2x_0^{'} + x_1^{'} + 5k = 2x_0 + x_1 \\ 0 \leq x_0, x_1 \leq 7 \\ 0 \leq x_0^{'}, x_1^{'} \leq 7 \\ x_0 \geq x_0^{'} + 1 \end{cases}$$

The points counting result is $C(\mathcal{P}_1) = \lfloor \frac{x_1}{N} \rfloor$ and

$$\begin{aligned} C(\mathcal{P}_0) = & \frac{8}{5} \times x_0 + [[0, \frac{2}{5}, -\frac{1}{5}, \frac{1}{5}, -\frac{2}{5}]_{x_0}, \\ & [0, -\frac{3}{5}, -\frac{1}{5}, \frac{1}{5}, -\frac{2}{5}]_{x_0}, [0, -\frac{3}{5}, -\frac{1}{5}, -\frac{4}{5}, -\frac{2}{5}]_{x_0}, \\ & [0, \frac{2}{5}, \frac{4}{5}, \frac{1}{5}, \frac{3}{5}]_{x_0}, [0, \frac{2}{5}, -\frac{1}{5}, \frac{1}{5}, \frac{3}{5}]_{x_0}]_{x_1}. \end{aligned}$$

$u(n) = [u_0, u_1, ..., u_{p-1}]_n$ is equal to the item whose rank is equal to $n \ mod \ p$ [11], thus

$$u(n) = \begin{cases} u_0, \ if \ n \ mod \ p = 0, \\ u_1, \ if \ n \ mod \ p = 1, \\ \cdots \\ u_{p-1}, \ if \ n \ mod \ p = p-1. \end{cases}$$

For $\vec{x} = (5,5)$, $C(\mathcal{P}_1) = \lfloor \frac{5}{5} \rfloor = 1$, $C(\mathcal{P}_0) = \frac{8}{5} \times 5 + 0 = 8$, $g(\vec{x}) = 8 + 1 = 9$.

5.2 Mapping to Hardware

By counting integer points in polytopes, we can generate an optimal intra-bank offset mapping function with the minimum storage overhead. However, as shown in Example 4, building the complex mapping function will cost too much in hardware resources. A trade-off between practicality and optimality is considered by using a memory-padding based heuristic approach. As shown in Fig. 4(b), padding two data elements on dimension x_1 insures that each row maintains two data elements of Bank 0. The calculation of the intra-bank offset for data (5,5) is simplified as $2 \times 5 + 1 = 11$.

Our heuristic method makes each polytope $\mathcal{P}_t$ have a constant number of data elements. Under lexicographic ordering, padding on dimension x_{d-1} is able to ensure the constant number of data elements in each $\mathcal{P}_t$. Take block-cyclic partitioning for example: the padding size q is calculated as

$$q = N \times B \times \lceil \frac{w_{d-1}}{N \times B} \rceil - w_{d-1}.$$

The points number in the padded polytope is calculated as

$$C(\mathcal{P}_t) = \begin{cases} \prod_{j=t+1}^{d-2} w_j \times \lceil \frac{w_{d-1}}{N \times B} \rceil \times B \times x_t & 0 \le t < d-2 \\ \lceil \frac{w_{d-1}}{N \times B} \rceil \times B \times x_t & t = d-2 \\ \lfloor \frac{x_t}{N \times B} \rfloor \times B + x_t \ mod \ B & t = d-1 \end{cases}$$

Example 5. Given the array reference $\vec{x} = (5, 5)$, the data domain $0 \le x_0, x_1 \le 7$, a partition vector $\vec{\alpha} = (2, 1)$, a partition factor $N = 5$, and partition block size $B = 1$, the intra-bank offset generated by padding is $C(\mathcal{P}_1) = 1$, $C(\mathcal{P}_0) = \lceil \frac{8}{5} \rceil \times 5 = 10$, $g(\vec{x}) = 10 + 1 = 11$.

6. PARTITIONING ALGORITHM AND DESIGN SPACE EXPLORATION

The design flow is shown as Fig 5. The algorithm's input is the iteration domain and memory references information extracted from the input code. The output is a selected partition strategy $(N, B, \vec{\alpha})$, which guides the code transformation. The main partitioning algorithm is composed of two parts: one part constructs the bank-mapping function and the other constructs the intra-bank offset function. The strategies are derived by bounded enumeration and conflict detection. The enumeration of N and $\vec{\alpha}$ is the same as the method in [21]. The enumeration of the block size starts from $B = 1$, and to reduce the resource resulting from the division operations, we only consider $B = 2^k$, $k \in \mathbb{Z}$, $k \ge 0$. A resource estimation model is built for selecting an optimal partition strategy. Assuming $B = 1$, the algorithm for cyclic partitioning is described in Algorithm 1.

Algorithm 1 Partitioning algorithm (block size fixed)

```
1: 𝕊 → partition strategy set
2:
3: 𝕊 = ∅
4: for N = m to ∏_{j=0}^{d-1} w_j do
5:     for each α⃗ in the space of N^d do
6:         empty=ConflictDetect(N, α⃗)
7:         if (empty) then add (N, α⃗) in 𝕊
8:     end for
9:     if (𝕊 ≠ ∅) break
10: end for
11: CalculatePadding(q)
12: Strategy → s ∈ 𝕊, minimize{ResourceEstimate(s, q)}
13: CodeGen(Strategy)
```

6.1 Resource Model

The partition strategy affects the resource usage of the hardware implementation of the transformed program. We introduce a resource estimation model and set up a standard for strategy selection. The use of resources is mainly made up of two parts: the storage and the address logic. We estimate the resource for a given partition strategy using a weighted arithmetic average, as Eqn. (6).

$$Est = c_0 \times U_{store} + c_1 \times U_{addr} \quad (6)$$

c_0 and c_1 reflect the importance of the resource, $c_0 + c_1 = 1$. U_{store} and U_{addr} are the percentages of BRAMs and address logic resource utilization on the target platform. In our method, the resource for array storage is block RAM. Registers and distributed memories can be alternatives, but currently we don't consider them. Ideally the use of block RAMs is only influenced by the partition factor N and the padding size q. The storage is calculated as

$$U_{store}(N, q) = \lceil \frac{(w_{d-1} + q) \times \prod_{j=0}^{d-2} w_j}{N \times BRAM_{size}} \rceil,$$

in which $BRAM_{size}$ is the block RAM's size of the target FPGA platform. However we find that the use of block RAMs in the implementation is also related to the data width and the port number P. Such dedicated estimation is platform dependent. Taking Virtex-7 from Xilinx for example, there are two block RAM modes: SDP (simple dual-port mode; one read, one write) and TDP (true dual-port; two read or write). The detailed relationship between the BRAM's mode and maximum port width is shown in Table 2. Using the method of partitioning for multi-port memory may not save BRAMs due to this relationship. Only when the width of the array is less than 18 bits, can two simultaneous accesses to one block RAM (RAMB18E1) in one cycle be achieved. The dedicated storage estimation for Virtex7 is formulated below, with N_P as the partition factor with port P ($P = 1$ or $P = 2$) and the $width$ of the target array.

$$U_{store-v7}(N_P, q, width) = \lceil \frac{(w_{d-1} + q) \times \prod_{j=0}^{d-2} w_j}{N_P \times \lceil \frac{width}{18} \rceil \times BRAM_{size}} \rceil.$$

U_{addr} estimates the resource for address generation. A template of code transformation after partitioning is given in Fig. 6. The address generation unit and data assignment unit are two pieces of code which will be repeated multiple times for generating the address logic for different references and banks. Assuming that the resource for the address generation unit is Ad_{gen} and for the data assignment unit is $Data_{assign}$, the total resource for the address logic is mainly estimated as

$$U_{addr}(N, B, \vec{\alpha}) = (Ad_{gen} + Data_{assign}) \times N \times m.$$

The value of N and B influences the physical resource for divisions and modulo operations. The value of $\vec{\alpha}$ affects the resource used for multiplications. For example, the resource for the implementation with $N = 4$, $B = 1$, $\vec{\alpha} = (1, 2)$ is less than using $N = 4$, $B = 3$, $\vec{\alpha} = (1, 3)$. As a result, we intend

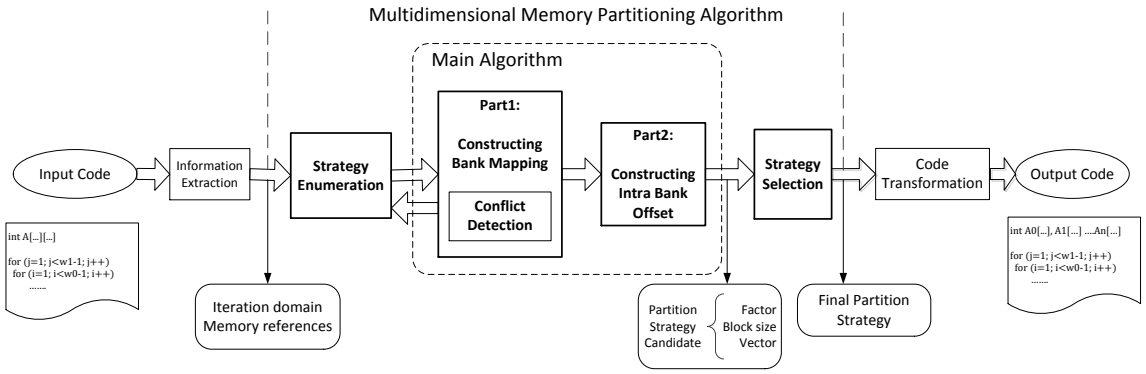


Figure 5: The design flow

Table 2: Virtex-7 BRAM mode and data width [1]

	Block RAM	Maximum port width	Simultaneous access
RAMB18E1(TDP)	1	18	2
RAMB18E1(SDP)	1	36	1
RAMB36E1(TDP)	2	36	2
RAMB36E1(SDP)	2	72	1

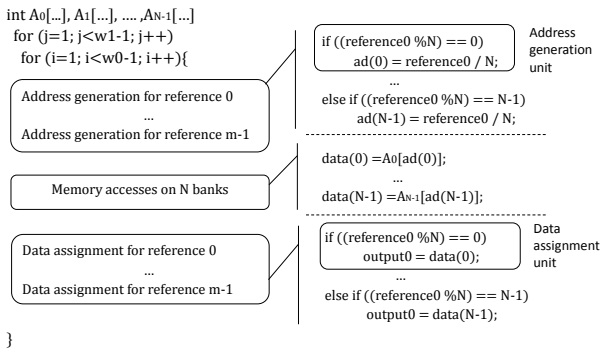


Figure 6: A template of code transformation

to use the number which is a power of two in the partition strategy.

7. EXPERIMENTAL RESULTS

The automatic memory-partitioning flow is implemented in C++ and is built in the open source compiler infrastructure ROSE [7]. ROSE is a flexible translator supporting source-to-source code transformation. We use Ehrhart testing provided by Polylib [6] for both polytopes emptiness testing and integer points counting, and Vivado Design Suite 2013.2 from Xilinx [9] as the high-level synthesis, logic synthesis, simulation, and power estimation tool. The RTL output targets the FPGA platform Xilinx Virtex-7. The high-level abstraction (C program) is parsed into the flow with the partition and loop pipelining directives. After memory-partitioning analysis and source-to-source code transformation, the transformed C code is synthesized into RTL through high-level synthesis and followed by logic synthesis, simulation, and power estimation tool.

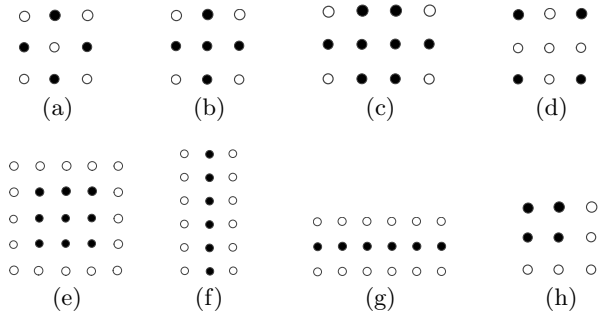


Figure 7: The access patterns of the benchmarks: (a) DENOISE (b) DECONV (c) DENOISE-UR (d) BICUBIC (e) SOBEL (f) MOTION-LV (g) MOTION-LH (h) MOTION-C

Eight loop kernels with different access patterns are selected from the real applications, such as medical image processing [14] and H.264 motion compensation [5], as shown in Fig. 7. In the experiments, we mainly focus on the effects brought by different access patterns.

The detailed experimental results are shown in Table 3, Table 4, and Table 5. We re-implemented the LTB method [21], and we compare the results to our GMP method. During the experiments, although we use exhaustive enumeration, the runtime for all the benchmarks is less than 1 second. Because all the ideal minimum partition factors are achieved, the enumeration space for the partition vector, which is related to the partition factor, is not large. We implement loop pipelining in the benchmarks and set the target throughput as II=1, which requires all of the memory accesses in the same iteration to be in one clock cycle. In the experiments, all of the two partitioning methods can achieve this target throughput requirement.

7.1 A Case Study: DECONV

We use benchmark DECONV as a case study on the effect of different intra-bank offset generation algorithms, as shown in Table 3 . Two algorithms are applied: optimal (integer points counting in polytopes), and heuristic (memory padding). Although the array size is increased by 4.69% with the heuristic method, it still uses the same number of

Table 3: Intra-bank offset generation comparison

	Array size (bits)	BRAM	Slice	DSP
Optimal	640*32	5	696	20
Heuristic	670*32	5	597	5
Compare	4.69%	0.00%	-14.22%	-75.00%

Table 4: Partition factor comparison

Partition Factor	BRAM	Slice	FF	DSP
Minimum	5	597	1298	5
Power-of-2	8	457	1279	0
Compare	60%	-23.45%	-1.46%	-100.00%

BRAMs as the optimal method. Because the size of the block RAMs on FPGAs is fixed, the small padding size did not increase the use of BRAMs. We tried 140 different array sizes, and compared to the array size, the padding rate is from 0.98% to 7%. As shown in the table, the heuristic method can reduce up to 14.22% slice and 75% DSP in this case. We used memory padding as our algorithm in the rest of our experiments.

We also had a case study on the partition factor. Two partition factors are applied: minimum (due to the conflict detection algorithm), and power-of-2 (increase the minimum partition factor to a number as power of two and also perform conflict detection). As shown in Table 4, although the use of BRAM is increased from 5 to 8, other resources are reduced. However, this is only suitable for the case when the power-of-2 partition factor is close to the minimum partition factor. As our main target in this paper is to reduce the bank number, we still use the minimum partition factor for DECONV, but use a power-of-2 partition factor under the multi-port partitioning mode.

7.2 Complete Experimental Results

Table 5 shows the complete experimental results on all eight benchmarks and the comparison with the LTB method in [21]. Information is presented in the table in the following order: access number in the inner-loop, data bitwidth, partition strategies, utilization of block RAMS, slices and DSPs, clock period, and dynamic power.

The LTB method only uses cyclic partitioning for single-port memory, while our GMP method can select a multi-port memory mode and use different block sizes. The bit-width of the array affects the selection of multi-port memory modes. According to the relationship between the port width and the block RAM's mode listed in Table 2, only when the width is less than 18 bit can two data elements from one RAMB18E1 under true dual-port mode be accessed simultaneously. As a result, three benchmarks (MOTION-LV, MOTION-LH and MOTION-C) with 8-bit width select the multi-port memory mode with $P = 2$. And for the block size in the partition strategy, as shown in the table, two benchmarks (DENOISE and BICUBIC) select the partition block size $B = 2$.

We set the access number in the inner-loop as the optimal partition factor for the single-port memory mode $P = 1$. With the selected partition strategies, our GMP method can achieve this target. Although LTB can already reduce the use of BRAMs and other resources in most of the cases, an optimal partitioning with a bank number equal to the access number is still not achieved in benchmark DENOISE and BICUBIC. However, with our GMP method using a block size two, all the bank numbers are reduced to the access number. Note that under the multi-port memory partitioning mode, we intend to choose a bank number that is a power of two. Thus, in MOTION-LV and MOTION-LH, despite $\frac{Access\#}{P} = 3$, we use four block RAMs as a trade-off.

The use of DSP is highly related to the division's number in the transformed code, which is related to the bank number. When the bank number is a power of two, DSP is reduced to zero. The divisions can be implemented as shift operators. Also, the use of slice can be reduced comparatively. Among the benchmarks, the bank number of DENOISE-UR and MOTION-C can already be reduced to a power of two by using LTB, while our GMP method can further reduce the use of DSP to zero on DENOISE, BICUBIC, MOTION-LV and MOTION-LH.

In all, compared to the previous work [21], our GMP method has an average reduction of 19.58% in BRAM, 20.26% in the use of slice, 50% in the number of DSP, and 10.09% in dynamic power.

8. CONCLUSION

In this work we propose a generalized memory partitioning (GMP) method using a polyhedral model. A theory based on polytope emptiness testing and integer points counting is presented for conflict detection between array references and intra-bank offset generation. To our knowledge, we are the first to use a polyhedral model to formulate and solve the bank access conflict problem. Experimental results demonstrate that, compared to the state-of-art partitioning algorithm, our proposed algorithm can reduce the number of block RAMs by 19.58%, slices by 20.26% and DSPs by 50%.

9. ACKNOWLEDGMENT

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 thank the UCLA/PKU Joint Research Institute and Xilinx for their support of our research.

10. REFERENCES

[1] *7 Series FPGAs Memory Resources.* http://www.xilinx.com/support/ documentation.
[2] *C-to-Silicon Compiler.* http://www.cadence.com/products/.
[3] *Catapult C.* http://calypto.com/.
[4] *Cynthesizer.* www.forteds.com.
[5] *JM Software.* http://iphome.hhi.de/suehring/tml/.
[6] *Polylib.* http://www.irisa.fr/polylib/.
[7] *ROSE compiler infrastructure.* http://rosecompiler.org/.
[8] *Synphony C Compiler.* http://www.synopsys.com.
[9] *Vivado High-Level Synthesis.* http://www.xilinx.com/.
[10] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data-parallel programs.

Table 5: Experimental results

Benchmark	Access #	Bit width	Method	BRAM	Slice	DSP	CP (ns)	Dynamic Power(mw)
DENOISE	4	32	LTB [21]	5	520	4	3.729	26
			GMP (P=1), B=2	4	303	0	3.395	16
			GMP vs LTB	-20.00%	-41.73%	-100.00%	-8.96%	-38.46%
DECONV	5	32	LTB [21]	5	597	5	4.538	27
			GMP (P=1), B=1	5	597	5	4.538	27
			GMP vs LTB	0.00%	0.00%	0.00%	0.00%	0.00%
DENOISE-UR	8	32	LTB [21]	8	794	0	3.738	31
			GMP (P=1), B=1	8	794	0	3.738	31
			GMP vs LTB	0.00%	0.00%	0.00%	0.00%	0.00%
BICUBIC	4	32	LTB [21]	5	483	4	4.364	24
			GMP (P=1), B=2	4	238	0	3.169	15
			GMP vs LTB	-20.00%	-50.72%	-100.00%	-27.38%	-37.50%
SOBEL	9	32	LTB [21]	9	1523	9	4.468	53
			GMP (P=1), B=1	9	1523	9	4.468	53
			GMP vs LTB	0.00%	0.00%	0.00%	0.00%	0.00%
MOTION-LV	6	8	LTB [21]	6	538	6	3.682	25
			GMP (P=2), B=1	4	425	0	3.169	25
			GMP vs LTB	-33.33%	-21.00%	-100.00%	-13.93%	0.00%
MOTION-LH	6	8	LTB [21]	6	536	6	3.946	21
			GMP (P=2), B=1	4	334	0	3.169	23
			GMP vs LTB	-33.33%	-37.69%	-100.00%	-19.69%	9.52%
MOTION-C	4	8	LTB [21]	4	174	0	3.405	14
			GMP (P=2), B=1	2	155	0	3.169	12
			GMP vs LTB	-50.00%	-10.92%	0.00%	-6.93%	-14.29%
Average			**GMP vs LTB**	-19.58%	-20.26%	-50.00%	-9.61%	-10.09%

Journal of Parallel and Distributed Computing, 26(1):72–84, 1995.

[11] P. Clauss and V. Loechner. Parametric parametric analysis of polyhedral iteration spaces. *Journal of VLSI Signal Processing*, 19(2):179–194, 1998.

[12] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transaction on Design Automation of Electronic Systems (TODAES)*, 16, 2011.

[13] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[14] J. Cong, V. Sarkar, G. Reinman, and A. Bui. Customizable domain-specific computing. *IEEE Design and Test of Computers*, 28(2):5–15, 2011.

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

[16] P. Li, L. N. Pouchet, D. Chen, and J. Cong. Loop transformations for throughput optimization in high-level synthesis. *In Proceedings of the ACM/SIGDA international Symposium on Field programmable gate arrays (to appear)*, 2014.

[17] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 488–495, 2012.

[18] Q. Liu, T. Todman, and W. Luk. Combining optimizations in automated low power design. In *Proc. of Design, Automation and Test Europe (DATE)*, pages 1791–1796, 2010.

[19] A. Schrijver. *Theory of Linear and Integer Programming*. JohnWiley and Sons, 1986.

[20] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 2007.

[21] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong. Memory partitioning for multidimensional arrays in high-level synthesis. *Proceedings of the 50th Annual Design Automation Conference (DAC)*, (12), 2013.

[22] Y. Wang, P. Zhang, X. Cheng, and J. Cong. An integrated and automated memory optimization flow for fpga behavioral synthesis. *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 257–262, 2012.

[23] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter AutoPilot: a platform-based ESL synthesis system. Springer, New York, NY, USA,, 2008.