PMP: Pattern Morphing-based Memory Partitioning in High-Level Synthesis

Dajiang Liu
College of Computer Science,
Chongqing University
liudj@cqu.edu.cn

Decai Pan
College of Computer Science,
Chongqing University
1164519853@qq.com

Xiao Xiong
College of Computer Science,
Chongqing University
202314021011@stu.cqu.edu.cn

Jiaxing Shang
College of Computer Science,
Chongqing University
shangjx@cqu.edu.cn

Shouyi Yin
School of Integrated Circuit,
Tsinghua University
yinsy@cqu.edu.cn

ABSTRACT

Memory partitioning is a widely used technique to reduce access conflicts on multi-bank memory in high-level synthesis. Previous memory partitioning methods mainly focus on a given access pattern extracted from stencil applications. Restricted by the pattern shape, these methods are prone to sub-optimal bank numbers or large overhead on address generation. In this work, we propose a pattern-morphing-based memory partitioning method, PMP, that only requires reduced hyperplane families to achieve the minimal bank number. To reduce the side effect of extra data padding, an integer linear programming problem is formulated for pattern morphing. Compared to the previous hyperplane-based memory partitioning, the experimental results show that our approach could achieve the optimal partition factor while saving 22% in LUTs, 21% in FlipFlops, 10% in DSPs, and 40% in memory overhead, on average.

KEYWORDS

High-Level Synthesis, Memory Partitioning, Pattern Morphing

ACM Reference Format:
Dajiang Liu, Decai Pan, Xiao Xiong, Jiaxing Shang, and Shouyi Yin. 2024. PMP: Pattern Morphing-based Memory Partitioning in High-Level Synthesis. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658239

1 INTRODUCTION

With a good balance of high performance, low power and short time-to-market, field-programmable gate arrays (FPGAs) have evolved into an attractive alternative to ASICs and general-purpose processors. Besides their traditional use, FPGAs are also used as hardware accelerators to speed up domain-specific applications, such as machine learning, computer vision, etc. A major challenge to implementing an FPGA accelerator is finding an efficient programming model. Traditional register-transfer level (RTL) design is time-consuming, error-prone and difficult to debug, and it takes weeks or months to implement a simple application.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3658239

High-level synthesis (HLS) is a breakthrough technology to simplify the programming of FPGA-based accelerators. By automatically transforming algorithms in C/C++ into RTL codes, HLS tools can significantly improve programming productivity and reduce the learning curve. Although existing commercial HLS tools are capable of generating high-quality RTL codes, there is still a great performance gap, as compared to manually optimized RTL circuits, even for stencil computations [9] with uniform reuse distances among array references. One important reason for this gap is the inefficient loop pipelining with constrained resources, e.g., limited on-chip memory port. In loop pipelining, the initiation interval (II) between adjacent iterations is the main concern. A smaller II indicates higher pipelining performance, yet incurs more simultaneous memory accesses. Typical block RAMs (BRAMs) in FPGA have limited ports to feed the highly parallelized computing units. When a loop includes a data array with multiple references, after loop pipelining, the number of simultaneous memory accesses of the array can easily exceed the port number of the BRAM. Duplicating data array into multiple copies can support simultaneous memory accesses, but it may lead to significant area and power overhead.

An alternative way is to partition the original data array into multiple memory banks such that multiple memory references are distributed to different memory banks, reducing the bank conflicts. Since each memory bank only holds a subset of the whole data array, memory partitioning results in much lower storage overhead, as compared to memory duplication. However, memory partitioning also incurs extra banking logic to identify the specific bank and the corresponding offset address for each memory reference. Moreover, restricted by the size and shape of the access pattern, it is difficult to guarantee a minimized bank number for memory partitioning. Thus, *how to minimize the number of memory banks with reduced hardware overhead is a challenging task in memory partitioning.*

Memory partitioning in HLS has been studied in several related works. The work in [1] attempts to partition and schedule multiple memory accesses of a data array in the same loop iteration to multiple banks in cyclic form. Then, the work in [4] schedules memory accesses in different loop iterations to find the near-optimal partitioning. Although these works take a first step towards efficient memory partitioning for loop pipelining, they are limited to one-dimensional data arrays, while many real-life applications often include nested loops with multi-dimensional arrays. The research in [9] presents a linear transformation-based (LTB) approach for multi-dimensional memory partitioning using linear hyperplane families. However, this method is limited to cyclic partitioning, which may lead to sub-optimal solutions in some cases. In [8], a generalized memory partitioning (GMP) method is proposed to

support block-cyclic partitioning on multi-dimensional arrays such that it could find the optimum in most cases. In [5], even data reuse is used to eliminate some memory references. In [2], a tessellation-based method is proposed with less hardware and energy consumption. In [10], a trace-based bank partitioning algorithm is proposed to support both affine and non-affine access patterns. In [3], a graph theory-based memory banking (GMB) method is proposed to achieve the optimal bank number. All previous methods, including hyperplane-based, lattice-based and graph-based, attempt to find the optimal bank number with a given shape of access pattern, losing opportunities to achieve both minimized bank number and reduced hardware overhead for a wide range of patterns.

In this paper, we propose an innovative Pattern-morphing-based Memory Partitioning (PMP) approach to achieve the minimized bank number with reduced hardware overhead. The key idea is to change the shape of a given access pattern by executing memory references from different iterations simultaneously; thus, only reduced hyperplane (with coefficients of 0 or 1) families are needed to separate these references to different banks. Our specific contributions are summarized as follows:

- A pattern-morphing-based multidimensional memory partitioning algorithm is proposed to simultaneously optimize the bank number and hardware resource. To the best of our knowledge, this is the first work to optimize memory partitioning by changing the shape of access patterns.
- An optimization problem using integer linear programming is established to find the morphing result with the minimal number of moving steps such that the side effects of data padding could be minimized.

2 BACKGROUND

In this section, we provide the definitions of the traditional hyperplane-based memory partitioning method since our work is built on it.

Definition 1. (Data Domain) [9] Given a finite n-dimensional data array A, the data domain $\mathcal{M}$ of the array is defined as a set of all memory elements. The coordinates of a data element $\vec{\phi} \in \mathcal{M}$ is expressed as $\vec{\phi} = (x_0, x_1, ..., x_{n-1})^T$, where $x_i \in [0, w_i - 1], 0 \le i < n$, and w_i indicates the width of the i^{th} dimension of the array.

Definition 2. (Access Pattern) [9] An access pattern consisting of P adjacent data references for an array A that needs to be simultaneously performed, which is represented as $\mathcal{P} = (\vec{\phi}_1, \vec{\phi}_2, \cdots, \vec{\phi}_P)$, where $\vec{\phi}_i = (x_0^i, x_1^i, ..., x_{n-1}^i)^T$.

Definition 3. (Memory Partitioning) [9] The memory partitioning of an array is represented as a pair of mapping functions $(f(\vec{\phi}), g(\vec{\phi}))$, where $f(\vec{\phi})$ maps the data to a memory bank and $g(\vec{\phi})$ generates the corresponding intra-bank offset.

Definition 4. (Valid Memory Partitioning) [9] A valid memory partition of a data array with data domain $\mathcal{M}$ is to find a pair of mapping functions $(f(\vec{\phi}), g(\vec{\phi}))$ such that:

$$\forall \vec{\phi}_i, \vec{\phi}_j \in \mathcal{M} : \ \vec{\phi}_i \neq \vec{\phi}_j \Leftrightarrow (f(\vec{\phi}_i), g(\vec{\phi}_i)) \neq (f(\vec{\phi}_j), g(\vec{\phi}_j)) \quad (1)$$

Definition 5: (Hyperplane-based Memory Partitioning) [8] Hyperplane-based memory partitioning is a block-cyclic partitioning scheme that involves a hyperplane or *partition vector* ($\vec{\alpha}$) and partition *block size* (B). For a 2-D data domain, its banking function

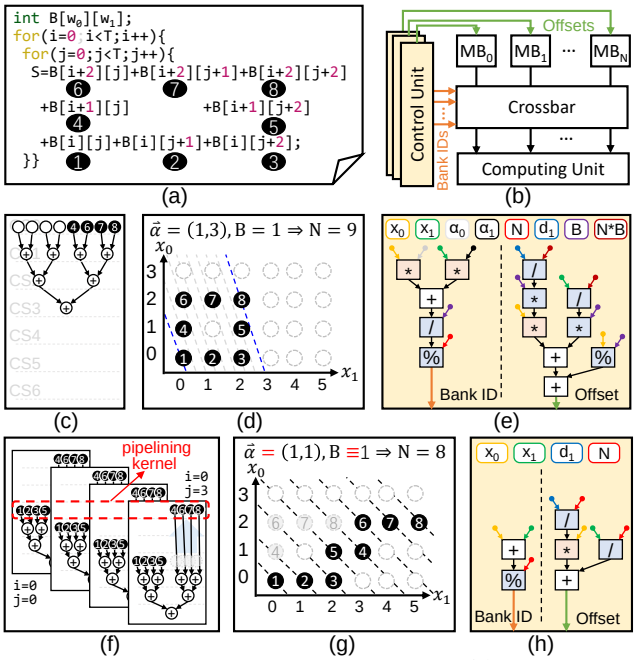

Figure 1: A motivating example of *prewitt*. (a) the *C* code of *prewitt*, (b) the hardware template for memory banking, (c) the original scheduling result, (d) the access pattern and partitioning result for the original scheduling, (e) the hardware overhead for the original scheme, (f) the new scheduling for new pipelining kernel, (g) the access pattern and partitioning result for the new scheduling, and (h) the reduced hardware overhead for the new scheme.

(f) and intra-bank offset function (g) could be described as follows:

$$f(\vec{\phi}) = \left\lfloor \frac{\vec{\alpha} \cdot \vec{\phi}}{B} \right\rfloor \%N = \left\lfloor \frac{(\alpha_0, \alpha_1) \cdot (x_0, x_1)}{B} \right\rfloor \%N \quad (2)$$

$$g(\vec{\phi}) = \left\lceil \frac{w_1}{N * B} \right\rceil * B * x_0 + \left\lfloor \frac{x_1}{N * B} \right\rfloor * B + x_1 \%B \quad (3)$$

To ensure a valid intra-bank address [8], the lower dimension of the data array should be an integer multiple of $N \times B$; thus, the *padding size*, q, for a 2-D array could be represented as:

$$q = N \times B \times \left\lceil \frac{w_1}{N \times B} \right\rceil - w_1 \quad (4)$$

Based on the above definitions, the main task of hyperplane-based memory partitioning is performing space exploration on ($\vec{\alpha}$, N, B) to make trade-offs among partition factor, storage overhead and addressing complexity. The addressing complexity could hardly be explicitly represented, yet it highly depends on the value of ($\vec{\alpha}$, N, B). For example, the power-of-two value could significantly reduce the complexity of multiplication, division and modulo operations in the banking function and offset function.

3 MOTIVATION AND CHALLENGE

In this section, we first present a motivating example to show the limitations of pattern-fixed memory partitioning and the core idea of our work. Then, the challenge of pattern-morphing-based memory partitioning is analyzed.

3.1 Motivating Example

Fig. 1(a) presents a loop kernel of a *prewitt* algorithm, which includes 8 memory references (❶ to ❽) of data array B, where w_0 and w_1 indicate the widths of the 0^{th} and 1^{th} dimension of the array.

Fig. 1(b) gives the hardware template for memory banking, where the control unit generates bank IDs and offsets for parallel data references from the computing unit. In memory partitioning, the number of used memory banks (MB), the size of each bank and the complexity of the control unit are the three main concerns.

Pattern-fixed memory partitioning. With naive scheduling on the extracted data-flow graph (DFG), as shown in Fig. 1(c), the 8 loading operations in the same iteration are performed simultaneously after fully pipelining (II=1). Correspondingly, as the data domain in Fig. 1(d) shows, we try to use perform memory partitioning with $N = 8$, $\vec{\alpha} = 1, 3$ and $B = 1$, yet it fails to get a conflict-free partitioning result. We found that, no matter what hyperplane family and block size we use, it always has at least one conflict with 8 memory banks; thus, we have to increase the partition factor to 9. To implement the corresponding banking function and offset function, as shown in Fig. 1(e), it at most includes 5 multiplications, 3 divisions, 2 modulo operations, 3 adders and 6 constants, leading to a complex circuit and a larger configuration buffer. Even if partition factor N and block size B are set to the power of two, i.e., some units (light blue boxes) could be simplified as shifters or AND gates, it still involves 3 multiplications (light orange boxes). Thus, pattern-fixed memory partitioning may neither find the optimal partition factor nor achieve a reduced control unit.

Pattern-morphing-based memory partitioning. To find the optimal partition factor with less overhead on the control unit, we introduce pattern-morphing-based memory partitioning. As shown in Fig. 1(f), the DFG is carefully scheduled, i.e., 4 load operations (❹,❻, ❼ and ❽) are moved 3 steps earlier. Consequently, as the new pipelining kernel shows, memory references from different iterations will be performed simultaneously. Correspondingly, as the data domain in Fig. 1(g) shows, the access pattern is morphed as a stepped shape. As a result, the optimal partition factor of 8 could be achieved using cyclic-only ($B = 1$) partitioning with a partition vector $\alpha = \vec{(}1, 1)$. Consequently, as depicted in Fig. 1(h), the corresponding banking function and offset function can be greatly simplified with only 6 operations, including 1 multiplier, 2 division, 1 modulo operation and 2 adders. When partition factor N is set to the power of two, our approach remains only one complex operation (multiplication) in address generation. Therefore, pattern-morphing-based memory partitioning makes it possible to find the optimal or near-optimal partitioning with a moderate overhead of the control unit.

3.2 Challenge for Pattern Morphing

So far, we have discussed the potential of pattern morphing-based memory partitioning. However, it also incurs data domain extension (data padding) and iteration domain extension. After pattern morphing, as shown in Fig. 1(f), memory references from different iterations are simultaneously performed, leading to the innermost loop carried dependencies. With loop-carried dependencies, the iteration corresponding to the edge of the data domain involves incomplete data for kernel computation. One of the solutions is to add some extra statements out of the innermost loop to load the missed data, yet it incurs an imperfectly nested loop that could hardly be flattened which would greatly decrease the pipelining performance. The other way is to pad dummy data at the edge of the data domain. If the maximal steps moved along the low dimension in pattern morphing is S, it will introduce loop-carried dependencies with the maximal distance of S, leading to $2S$ data padding at the lower dimension of the data array and additional

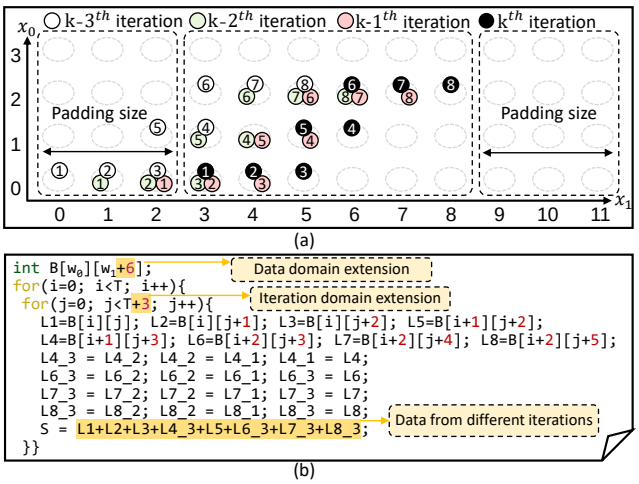

Figure 2: The challenge of patter-morphing-based memory partitioning. (a) Data padding in the data domain, and (b) the changed source code.

S iterations of the innermost loop. Fig. 2 gives the data padding result for the loop in the motivating example. Since ❹, ❻, ❼ and ❽ move 3 three steps in the data domain, it will produce 4 loop carried dependence with a distance of 3. Consequently, as shown in Fig. 2(a), the lower dimension of the data domain is expanded to $w_1 + 6$. At the current (k^{th}) iteration, the result data of ④, ⑥,⑦ and ⑧ from the k-3^{th} iterations and the result of ❶, ❷, ❸ and ❺ from current (k^{th}) iterations are all ready for kernel computation. Consequently, it also generates a new source code with an expanded iteration domain, as depicted in Fig. 2(b). From this example, we find that memory partition with the minimal partition factor and reduced partition vector may require large moving steps in pattern morphing, incurring a large overhead from data padding. Thus, *how to get the optimal partition factor with a simplified hyperplane family while keeping a moderate number of moving steps in pattern morphing is a challenging task in pattern morphing-based memory partitioning.*

4 FORMULATION OF PATTERN MORPHING

Given a hyperplane family, a bank number and a target II, how to perform pattern morphing is crucial to reduce the storage overhead caused by data padding. Thus, we first formulate the sub-problem of storage minimization into an integer programming problem (ILP). Then, the bank number and hyperplane complexity are explored in an iterative fashion.

In the sub-problem, we assume that the morphed pattern is handled by hyperplane-based memory partitioning with cyclic strategy only. In other words, block partitioning is neglected, i.e., $B = 1$, since the search space extended by adding pattern morphing could already offer the optimal partition factor in most cases. Before formulation, we first define some global parameters for the sub-problem as follows:

4.1 Definitions

- P: The number of references in a pattern $\mathcal{P}$.
- X_p: The offset of reference p of P along the x_1 axis in the data domain.
- Y_p: The offset of reference p of P along the x_0 axis in the data domain.
- N: The given number of memory banks.
- II: The target initiation interval for loop pipelining.

- (α_1, α_2): the vector for the partition hyperplane family.

Based on the parameters above, we define 5 variable sets for ILP formulation. These variable sets are as follows:

- st_p: The moving steps along x_1 axis for reference p in $\mathcal{P}$, which is an integer with a range of $[0, P]$. As moving steps along higher dimensions (e.g., x_0) would cause long-distance data reuse that involves high resource consumption, the pattern morphing only focuses on the lowest dimension (x_1).
- $maxStep$: The maximal number of moving steps over all references in $\mathcal{P}$.
- bk_p: The reference p's bank ID, which is a non-negative integer with a range of $[0, N)$.
- $b_{p,n}$: The reference p is assigned to the n^{th} bank or not, which is 0-1 binary variable.
- $ax0_p$: The integer auxiliary variable for linear representation of bk_p, which is a non-negative integer.
- $ax1_{p,n}$: The 0-1 auxiliary variable for non-overlapped movement constraint.
- $ax2_{p,n}$, $ax3_{p,n}$: The 0-1 auxiliary variables for linear representation of $b_{p,n}$.

Padding size: Based on the maximal moving step, the padding size of pattern morphing-based memory partitioning could be deduced from equation (4) and becomes:

$$q = N \times \left\lceil \frac{w_1 + 2 \times maxStep}{N} \right\rceil - (w_1 + 2 \times maxStep) \tag{5}$$

4.2 Constraints

The sub-problem of storage minimization involves 5 basic constraints, which are described one by one in the following part.

C1: Banking function in integer form. By setting the blocking factor as 1, the banking function in (5) could be represented as follows:

$$\forall p \in [0, P) : bk_p = (\alpha_0 \cdot Y_p + \alpha_1 \cdot X_p)\%N \tag{6}$$

With the help of integer auxiliary variables, $ax0_p$, the nonlinear modulo operation could be further simplified as follows:

$$\begin{aligned} 0 \le bk_p < N \\ bk_p = (\alpha_0 \cdot Y_p + \alpha_1 \cdot X_p) - ax0_p \cdot N \end{aligned} \tag{7}$$

C2: Moving steps constraints. For each element, its moving steps must be less than or equal to the maximal steps, which are represented as follows:

$$\forall p \in [0, P) : st_p \le maxStep \tag{8}$$

C3: Non-overlapped movements. For any two references in $\mathcal{P}$, their new positions must not be overlapped after movements. This constraint could be described as follows:

$$\forall p, q \in [0, P), p \ne q : \ (w_1 \cdot Y_p + X_p) - (w_1 \cdot Y_q + X_q) \ne 0 \tag{9}$$

By introducing a 0-1 auxiliary variable, $ax1_{p,q}$, the above inequality could be further simplified as follows:

$$\begin{aligned} w_1 \cdot Y_p + X_p - w_1 \cdot Y_q + X_q < ax1_{p,q} \cdot \Omega_0 \\ w_1 \cdot Y_p + X_p - w_1 \cdot Y_q + X_q > -(1 - ax1_{p,q}) \cdot \Omega_0 \end{aligned} \tag{10}$$

where Ω_0 is a big enough positive constant integer and setting it to the array size $(w_1 \times w_0)$ would work well.

C4: Banking function in binary form. Based on the banking function in integer form, we can conduct the banking function in binary form as follows:

$$\forall p \in [0, P), n \in [0, N) : b_{p,n} = \begin{cases} 1, & if\ bk_p - n = 0 \\ 0, & Otherwise \end{cases} \tag{11}$$

By introducing two 0-1 auxiliary variables $ax2_{p,n}$ and $ax3_{p,n}$, the above equation could be linearlized as follows:

$$\begin{aligned} b_{p,n} + ax2_{p,n} + ax3_{p,n} = 1 \\ (bk_p - n) - \Omega_1 \cdot ax2_{p,n} + ax3_{p,n} \le 0 \\ (bk_p - n) + \Omega_1 \cdot ax3_{p,n} - ax2_{p,n} \ge 0 \end{aligned} \tag{12}$$

where Ω_1 is also a big enough positive constant integer and we set it to the array size $(w_1 \times w_0)$.

C5: Conflict-free bank allocation. With a given II and bank number N, the conflict-free constraint can be represented as follows:

$$\forall n \in [0, N) : \sum_{p \in [0,P)} b_{p,n} \le II \tag{13}$$

4.3 Objective Function

Based on the above constraints, the objective is to minimize both the maximal moving step and the total number of moving steps, which is described as follows:

$$\min \Omega_2 \cdot maxStep + \sum_{p \in [0,P)} st_p \tag{14}$$

where Ω_2, a big enough integer constant, indicates that minimizing the maximal moving step is of higher priority. With the second item in equation (14), a smaller number of total moving steps could be achieved, which could not only reduce the complexity of code transformation but also reduce the consumption of FILP-FLOPs for data delivery. It is generally known that ILP has poor scalability. However, as the sizes for most access patterns are relatively small (several to dozens of references), the formulated ILP problem is still of moderate complexity.

5 PMP METHOD

So far, we have only solved the sub-problem of padding size optimization. In this section, we will further explore the partition factor and hyperplane family in an iterative search manner. Then, we present the template for code transformation.

5.1 Iterative search algorithm

Algorithm 1 presents the iterative search process for the optimized partition factor and reduced hyperplane family. It takes the access pattern $\mathcal{P}$ and II as inputs, and takes partition factor (N), hyperplane family (α), and morphed pattern ($\mathcal{P}^*$) as outputs. At the beginning, it calculates the minimal bank number in theory (line 1). Then, it constructs a set of reduced hyperplane families C, i.e., {(0,1),

Algorithm 1: Iterative search for partition factor and vector

Input: an access pattern $\mathcal{P}$ with P references and II.
Output: Bank number N, hyperplane family c, new pattern $\mathcal{P}^*$

1 $N \leftarrow \lceil P/II \rceil$;
2 $C \leftarrow constructReducedHyperplanes$;
3 **while** *true* **do**
4 **for** *each hyperplane family* $\vec{\alpha} \in C$ **do**
5 $isSucc, padSize, \mathcal{P}^* \leftarrow ILPSolve(\mathcal{P}, N, \vec{\alpha}, II)$;
6 **if** $isSucc == true\ and\ padSize < threshold$ **then**
7 $return\ N,\ padSize,\ \mathcal{P}^*$;
8 **end**
9 **end**
10 $N \leftarrow N + 1$
11 **end**

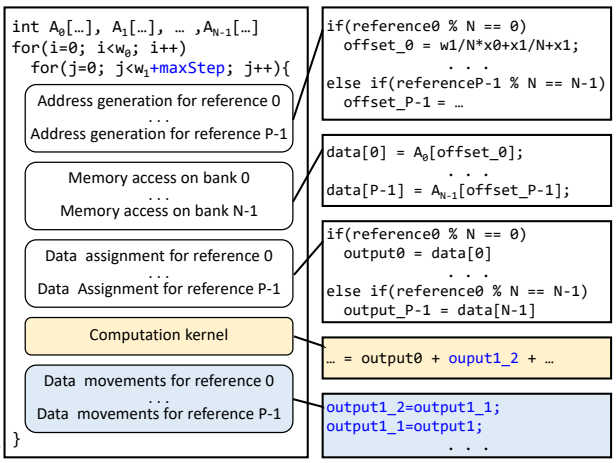

Figure 3: The template for code transformation.

(1,1), ···}. With coefficients of the reduced hyperplane, the corresponding control unit for banking functions and offset functions could be significantly reduced. Next, the algorithm enters a while loop to search for the partition factor (lines 3-10). For each partition factor (N) and each hyperplane $\vec{\alpha}$, it solves the constructed ILP problem to get the padding size-minimized pattern. If it succeeds and the padding size is less than a threshold, it completes the search process and returns the partition factor and new pattern. Otherwise, it will try the next reduced hyperplane family. If the whole hyperplane family set is traversed and there is still no satisfactory solution, it will increase the required partition factor by one and restart from the beginning.

Since the partition factor is searched from the minimal one while the hyperplane is searched from the simplest one, our work could find a solution with an optimized partition factor and simplified control unit for memory banking. Moreover, as the pattern morphing is solved by ILP formulation, our work could also keep a moderate overhead from data padding.

5.2 Template for Code Transformation

After the pattern morphing result is obtained, we will conduct source-to-source code transformation. Fig. 3 presents the template for code transformation, which has 5 parts in the loop body, including address generation, memory access, data assignment, computation kernel and data movement. Compared to the template of traditional hyperplane-based memory partitioning, our template has 3 differences: 1) the trip count of the innermost loop is extended by adding *maxStep* iterations; 2) the data in the computation kernel may include results from previous iterations. As shown in Fig. 3, the result *output*1_2 is from the load result before two iterations; 3) a data movement part is added to get the result from previous iterations.

6 EXPERIMENTAL RESULTS

6.1 Experimental Setup

The automatic flow of PMP is implemented in C++ and built on an open-source compiler infrastructure ROSE [7], which is a flexible translator supporting source-to-source code transformation. We use GLPK [6] to solve the ILP problem and Vivado Design Suite 2018.3 from Xilinx as the tools for high-level synthesis, logical synthesis, simulation and power estimation. The target FPGA device for RTL implementation is XC7K160tffg676-3 Kintex-7. The C programs of kernels are parsed into the flow with loop pipelining and array

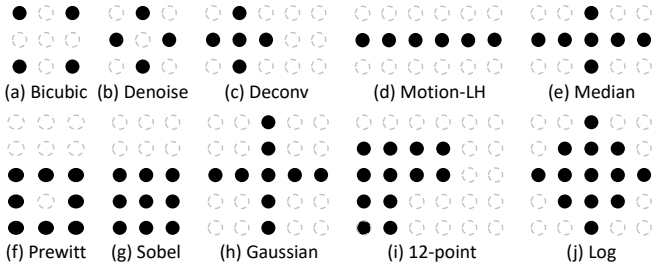

Figure 4: The access patterns of benchmarks.

partitioning directives. With only 3 reduced hyperplane families {(0,1), (1,0), (1,1)}, PMP performs memory-partitioning analysis and source-to-source code transformation. Then, the transformed C code is synthesized into Verilog RTL through high-level synthesis and followed by logic synthesis, simulation, and power estimation tool.

As depicted in Fig. 4, 10 kernels with different access patterns are selected from real applications, such as image processing and H.264 motion compensation. The dimension sizes for the input 2-D arrays of these kernels are all set to 100 and the IIs for all kernels are set to 1. Since GMP[8] and graph-based memory banking (GMB) [3] are the two state-of-the-art methods for parallel data access on a single data array, we re-implemented both methods for the same set of benchmarks for comparison. In GMB, it first extends the original pattern into an extended stencil graph (ESG), performs graph coloring on the ESG, and determines a repeated super-tile for lookup-style banking.

6.2 Resource, Timing and Power Comparison

Table 1 presents the experimental results of resource, timing and power. The parameters (#1,#2),#3 for GMP indicate the partition factor and partition block size while those of PMP indicate the partition vector and the maximal moving steps. The parameter (#1,#2) for GMB indicates the width and the length of the super-tile.

Comparison with GMP. On overage, our method can achieve an average reduction of 2% in bank number, 22% in LUTs, 21% in FFs, 10% in DSPs, 18% in clock period and 4% in power consumption. The reduction in bank numbers mainly comes from the morphed pattern. For the kernel of *Prewitt* and 12-*point*, GMP can not get the minimal partition factor with the given shape of the patterns; while PMP can use a reduced hyperplane (1,1) to perform partitioning after the pattern shape is morphed. The main reason for reductions in LUTs, FFs and DSPs is the reduced hyperplane family. As the parameter column shown in table 1, PMP mainly used (1,1) or (0,1) hyperplane while GMP used complicated hyperplane as well as block size, leading to complex banking and offset functions and more resource consumption. However, there are exceptions on the kernels of 12-*point* and *Log*, where PMP consumes more LUTs and FFs than those of GMP; the reason is that PMP gets relatively large moving steps (5 and 8) in pattern morphing, leading to more resources to deliver loop carried dependences. Overall, PMP can outperform GMP in partition factor, resource consumption, and clock period in most cases.

Comparison with GMB. On overage, our method can achieve an average reduction of 2% in bank number, -7% in LUTs, 45% in FFs, 10% in DSPs, -22% in clock period and 8% in power consumption. Ours can achieve the optimal bank number over all kernels while GMB failed to get the optimal one on *Prewitt* and *Gaussian*. For these 2 kernels, GMB cannot form perfect extended stencil graphs,

Table 1: Resource usage, timing and power comparison

Kernel	Method	#Bank	LUT	FF	DSP	CP[1]	Pow.[2]	Para.
Bicubic	GMP	4	268	48	0	1.9	151	(1,2),1
	GMB	4	365	336	0	1.9	129	(4,4)
	PMP	4	226	48	0	1.8	144	(1,1),1
Denoise	GMP	4	296	48	0	1.8	153	(1,3),2
	GMB	4	348	333	0	1.8	129	(4,4)
	PMP	4	236	48	0	1.8	145	(1,1),1
Deconv	GMP	5	1371	943	0	4.4	170	(4,3),2
	GMB	5	898	926	0	1.8	199	(5,5)
	PMP	5	817	469	0	2.4	184	(1,1),2
Motion-H	GMP	6	1212	651	0	3.6	206	(1,1),1
	GMB	6	624	834	0	2.2	177	(1,6)
	PMP	6	1270	651	0	2.4	214	(1,0),0
Median	GMP	7	1682	1121	0	3.3	195	(1,2),1
	GMB	7	1439	1507	0	2	237	(7,7)
	PMP	7	1523	911	0	3	214	(1,1),2
Prewitt	GMP	9	2841	1783	0	2.9	287	(1,3),1
	GMB	9	1182	1237	0	1.7	322	(3,3)
	PMP	8	622	111	0	2.3	160	(1,1),3
Sobel	GMP	9	3166	1978	0	2.9	308	(1,3),1
	GMB	9	1159	1403	0	2.1	238	(3,2)
	PMP	9	1824	1183	0	2.8	234	(1,1),4
Gaussian	GMP	9	3572	2461	9	8.1	214	(3,8),3
	GMB	10	2025	1802	0	1.9	319	(10,10)
	PMP	9	1951	1231	0	3.1	226	(1,1),4
12-point	GMP	13	2329	1354	0	4.4	230	(7,10),4
	GMB	12	2246	1597	0	3	235	(6,6)
	PMP	12	2680	1480	0	3.3	257	(1,1),5
Log	GMP	13	2535	1274	0	3.2	274	(1,5),1
	GMB	13	2516	2051	2	3.6	290	(13,13)
	PMP	13	2921	1681	0	3.2	260	(1,1),8
Average Improv.	vs GMP	2%	22%	21%	10%	18%	4%	-
	vs GMB	2%	-7%	45%	10%	-22%	8%	-

1. clock period (ns); 2. power (mW).

leading to extra references for graph coloring. As GMB offers two intermediate memories with the size of the super-tile for looking up the bank ID and offset address, it consumes much fewer LUTs for banking and offset functions and results in a smaller clock period; however, intermediate memories inevitably incur much overhead in FFs. Overall, PMP can outperform GMB in partition factor and FFs in most cases.

6.3 Memory Overhead and Running Time

Fig. 5 presents the running time and the memory overhead caused by data padding. On overage, PMP has 40.1% and 34.1% reduction in memory overhead, as compared to GMP and GMB, respectively. As presented in equation (5), although $2 \times maxStep$ elements need to be padded to the lower dimension for pattern morphing, the final size of the lower dimension only needs to be an integer multiple of N, rather than the integer multiple of $N \times B$ for GMP in equation (4). Thus, we can still outperform GMP in memory overhead in most cases. As to GMB, it needs to pad all dimensions of the array to an integer multiple of the super-tile size, leading to notable overhead when the super-tile becomes large. As the curves shown in Fig. 5, PMP consumes 48 × running time of GMP yet 17% time of GMB,

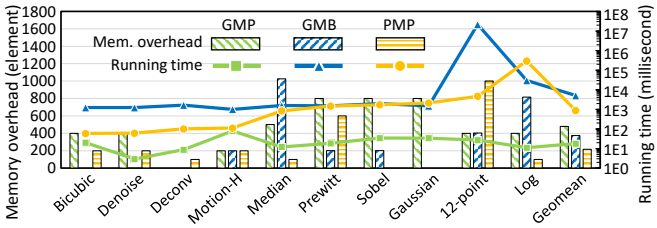

Figure 5: Memory overhead and running time comparison.

on geometric mean. As GMB needs to conduct graph coloring on an extended stencil graph that is much bigger than the pattern size, it takes the most running time. Whereas the ILP problem of PMP is related to the pattern size, it has a moderate running time, less than 5 seconds in most cases. Inevitably, with the maximal moving steps shown in Table 1, PMP also has a slight increase in execution latency from iteration extension; however, with the increase of loop size, this latency becomes negligible.

7 CONCLUSION

Memory partitioning is a key technique to increase data access parallelism. Existing methods mainly focus on a given shape of the access pattern, losing opportunities to achieve both minimized partition factor and significantly reduced resource consumption. To this end, this paper introduces pattern morphing to memory partitioning such that it only needs reduced hyperplane to achieve the minimal partition factor. To reduce memory overhead, we formulate the pattern morphing into an ILP problem. The experimental results show that our approach could achieve a good balance.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant 62274019, and the National Key Research and Development Project of China (No. 2020AAA0104603). The corresponding author of this paper is Dajiang Liu (liudj@cqu.edu.cn).

REFERENCES

[1] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic memory partitioning and scheduling for throughput and power optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 16, 2 (2011), 1–25.

[2] Juan Escobedo and Mingjie Lin. 2016. Tessellation-based multi-block memory mapping scheme for high-level synthesis with FPGA. In *2016 International Conference on Field-Programmable Technology (FPT)*. 125–132.

[3] Juan Escobedo and Mingjie Lin. 2018. Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2018).

[4] Peng Li, Yuxin Wang, Peng Zhang, Guojie Luo, Tao Wang, and Jason Cong. 2012. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *Proceedings of the international conference on computer-aided design*. 488–495.

[5] Binbin Liu, Fan Yang, Dian Zhou, and Xuan Zeng. 2020. An Efficient Memory Partitioning Approach for Multi-Pattern Data Access in STT-RAM. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–4.

[6] Andrew Makhorin. 2008. GLPK (GNU linear programming kit). *http://www. gnu. org/s/glpk/glpk. html* (2008).

[7] Dan Quinlan and Chunhua Liao. 2011. The ROSE Source-to-Source Compiler Infrastructure. http://rosecompiler.org/. In *in Cetus users and compiler infrastructure workshop, in conjunction with PACT*.

[8] Yuxin Wang, Peng Li, and Jason Cong. 2014. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. 199–208.

[9] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*. 1–8.

[10] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. 2017. A New Approach to Automatic Memory Banking using Trace-Based Address Mining. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2017).