IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 29, 2017, accepted January 18, 2018, date of publication January 26, 2018, date of current version March 12, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2798586

Multi-Bank Memory Aware Force Directed Scheduling for High-Level Synthesis

SHOUYI YIN, TIANYI LU, XIANQING YAO, ZHICONG XIE, LEIBO LIU, AND SHAOJUN WEI
Institute of Microelectronics, Tsinghua University, Beijing 100084, China

Corresponding author: Tianyi Lu (lodge671@foxmail.com)

ABSTRACT High-level synthesis has been widely recognized and accepted as an efficient compilation process targeting field-programmable gate arrays for algorithm evaluation and product prototyping. However, the massively parallel memory access demands and the extremely expensive cost of single-bank memory with multi-port have impeded loop pipelining performance. Thus, based on an alternative multi-bank memory architecture, a joint approach that employs memory-aware force directed scheduling and multi-cycle memory partitioning is formally proposed to achieve legitimate pipelining kernel and valid bank mapping with less resource consumption and optimal pipelining performance. The experimental results over a variety of benchmarks show that our approach can achieve the optimal pipelining performance and meanwhile reduce the number of multiple independent memory banks by 49.2% on average, compared with the state-of-the-art approaches.

INDEX TERMS Modulo scheduling, memory partitioning, multi-bank memory, HLS.

I. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) have earned a significant market segment in microelectronics industry. Off-the-shelf FPGAs are extensively available and FPGA-based accelerators have shown computation and energy efficiency in many application areas, in contrast with CPU/GPU [1]. High-Level Synthesis (HLS) is an efficient compilation process targeting FPGAs for algorithm evaluation and product prototyping. With high level description language (e.g. C program) as input, HLS can produce optimized hardware description language (HDL) output which can be synthesized into FPGAs. Thus, the design and verification efforts of digital systems can be dramatically reduced [2].

In HLS, optimizing loops performance is crucial for the overall performance of synthesized architecture and circuits. Loop transformation and software pipelining are commonly used to improve the parallelism of loop iterations at iteration level and instruction level respectively.

The loop transformations are commonly formulated as iteration-level polyhedral transformations [3]–[5]. There are several basic transformations at iteration-level, including fusion/fission, interchange, reverse, skewing, peeling, index set splitting and tiling. Software pipelining is a prominent technique of loop optimizations focusing on instruction-level parallelism [6], which is realized by overlapping the execution of subsequent iterations to parallelize loop execution. Modulo scheduling is arguably the most widely used technique to enable software pipelining [6], [7]. The key of modulo scheduling is to find a legitimate pipelining kernel with minimized Initiation Interval (II) by overlapping consecutive iterations. II is the delay between the initiation of two consecutive iterations of a loop and is inversely proportional to the pipelining performance [6]. After overlapping, operations are assigned to a particular control step and the scheduled operations will form a kernel. The remaining tasks of HLS are allocation and binding for specifying hardware resources consumption and mapping the operations to hardware units, respectively.

It is worth noting that after loop transformation and software pipelining, the loop parallelism is greatly improved, which also means that more concurrent memory accesses are required. Even if the pipelining is constituted successfully, the memory access conflicts will still cause serious degradation in pipelining performance. Therefore, massively parallel memory accesses with limited memory ports are becoming a crucial bottleneck for loop execution.

This memory access bottleneck requires a memory infrastructure which can be accessed simultaneously by parallel load/store operations. However, it is very difficult to provide all the necessary ports in single-bank data memory, since the extremely expensive cost of single-bank memory with multi-port in terms of area, power, and speed [8]. Typical

BRAMs in commonly available FPGAs are designed as single or dual port, which is very limited for feeding the highly parallelized memory accesses. An alternative architecture of multiple independent memory banks (multi-bank) is favored increasingly. Meanwhile, BRAM is configurable and flexible for programming, which provides good support for multi-bank memory architecture to relieve the bottleneck problem. In multi-bank memory architecture, data elements are mapped into multiple memory banks, so as to improve the bandwidth of parallel data access and reduce memory access conflicts. However, multi-bank memory only provides an architectural basis for solving the parallel data access problem, while an efficient pipelining and memory partitioning approach for multi-bank memory is still required.

Recently, several linear transformation based techniques [9]–[12] are proposed to partition data arrays into multi-bank memory for HLS. Nevertheless, multi-cycle memory access pattern in pipelining kernel is not concerned, which can cause considerable cost in memory bank consumption. Moreover, the interaction between modulo scheduling and memory partitioning is not considered, which can cause unnecessary memory access conflicts or waste of local memory banks. Actually modulo scheduling can further optimize memory access pattern, reducing the demand of parallel data accesses in pipelining kernel. Most of the existing HLS scheduling techniques only concern the scheduling problem under timing or hardware constraints [13], [14]. These shortcomings cause the limitation of the state-of-the-art approaches in loop pipelining. Efficient solution amenable to the emerging memory access constraints is demanded for HLS scheduling.

In this paper, a joint approach, which simultaneously considers modulo scheduling and memory partitioning for multi-bank memory, is formally proposed. In particular, our main contributions are described as follows:

- **A multi-cycle memory partitioning (MMP) algorithm for constructing conflict-free memory access pattern with minimized memory bank number.** The multi-cycle memory partitioning algorithm is employed to eliminate the memory access conflicts in each control step and all the load/store data elements in each cycle in the pipelining kernel can be accessed without any bank-conflict.
- **A memory-aware force directed scheduling (MAFDS) algorithm for operation scheduling in modulo scheduling.** Our method heuristically assigns operations distribution and constructs pipelining kernel by successively and iteratively scrutinizing the memory-aware force until a legitimate loop pipelining with minimized *II* can be found. With the memory-aware force, the memory access pattern in pipelining kernel is concurrently optimized to interact with memory partitioning for further reducing memory bank number.
- **A joint approach consisting of operation scheduling and memory partitioning for optimizing loop acceleration in HLS by reducing memory access constraints in the context of multi-bank memory.** From the perspective of loop optimization, our proposed approach is complementary to existing loop transformation techniques. Compared with the state-of-the-art approaches, the proposed joint approach can produce substantial reduction in the number of multiple independent memory banks, while achieving the optimal loop pipelining performance.

The remainder of this paper is organized as follows: In Section II, the related works are presented. Section III gives some notations and presents a motivation example. Then, Section IV describes the problem formulation. Section V gives an efficient solution of the joint approach. Next, Section VI gives the experimental results and comparisons that demonstrate the effectiveness of our approach. At last, Section VII summarizes conclusions.

II. RELATED WORK

Loop transformation and software pipelining are commonly performed by high-level FPGA compilers to improve the parallelism of loop iterations at iteration level and instruction level respectively.

Morvan *et al.* [3] propose a method using polyhedral analysis to improve nested loop pipelining. To overcome conflicts of memory dependencies in a pipeline, their approach firstly flattens the nested loop and then inserts wait states to resolve memory conflicts. Zuo *et al.* [4] present an integrated technique using polyhedral models to model and enable both intra- and inter-block optimizations, which improves the opportunity to use HLS optimizations for parallelism and pipelining. Liu *et al.* [5] propose a loop splitting technique, which is realized based on polyhedral analysis and transformation. For a given loop, it can resolve all potential conflicts of uncertain or non-uniform memory dependency at compile time.

In addition to these aforementioned iteration-level loop transformations, our approach targets instruction-level optimization for modulo scheduling in HLS. Early in high-level synthesis, the central research topic was the operation/instruction level scheduling [14]. Scheduling assigns operations in the Data Flow Graph (DFG) to the control steps and decides in which clock cycle they are performed. Various approaches have been proposed [13]–[15] for HLS scheduling. Force directed scheduling (FDS) [15] is a widely used constructive heuristic, with high flexibility and low complexity. It leverages the force, which is much like the force exerted by a spring that obeys Hooke's law, to characterize the assignment of an operation to a particular control step and proceeds by stepwise refinement. System of Difference Constraints (SDC) [13] uses a mathematical framework to describe scheduling constraints and performs optimization by solving a linear programming (LP) problem.

In the compiler domain, modulo scheduling is a well-known technique to exploit parallelism between successive iterations of a loop. Since timing and hardware resources constraints have made modulo scheduling an NP-hard problem [16], various heuristics have been proposed. Iterative modulo scheduling [17] combines list-scheduling, backtracking and a modulo reservation table to find the possible minimized *II* under the given constraints. Swing modulo scheduling [18] schedules the operations by considering the criticality and places each operation close to either its predecessors or successors to reduce the register requirements. Recently, a heuristic in [19] extend SDC [13] to handle modulo scheduling. This work prioritizes operations that minimize the impact on unscheduled operations and chooses the operations to be scheduled greedily. [20] also extends SDC and takes an alternative backtracking approach.

One major weakness of the existing modulo scheduling techniques is lack of awareness of memory access pattern optimization in pipelining kernel. With the support of multi-bank memory architecture, memory partitioning can improve the bandwidth of parallel data access and reduce memory access conflicts. With optimized memory access pattern, it can further reduce the number of multiple independent memory banks. In [9], a Linear Transformation-Based (LTB) method is proposed for data access in multidimensional arrays by cyclic partitioning. The same authors extend their previous work as a Generalized Memory Partitioning (GMP) algorithm in [10], using polyhedral model to describe the partitioning problem. A constructive algorithm (EMP), proposed in [11], develops an efficient memory partitioning strategy with low time complexity and low storage overhead for multidimensional array partitioning. Su *et al.* extend LTB to develop an efficient algorithm (DRMP [12]) for memory partitioning by caching the reusable data by on-chip registers. However, very few studies have extended the aforementioned state-of-the-art approaches to support memory partitioning in multi-cycle pipelining kernel. In [21], we have proposed an early version of joint loop mapping and data placement approach for coarse-grained reconfigurable architecture (CGRA). However, in CGRA, the loops are temporally mapped onto processing elements (PE), in which time multiplexing of PE is allowed. Therefore the operator scheduling is quite different to the spatial mapping in FPGA. In this paper, we constructively propose a joint modulo scheduling and memory partitioning scheme especially for FPGA.

III. MOTIVATION EXAMPLE

The front-end of HLS is similar to the existing software compiler. In the case of C as input for behavioral description, it involves lexical and syntactic analysis, and generates the corresponding intermediate representation (IR) such as a graph representation called Data Flow Graph (DFG). A DFG $G_d = (V_d, E_d)$ is a directed graph, where V_d is the set of operation nodes and E_d includes the data dependencies between operation nodes. We assume that all the operations in V_d incur one clock cycle delay and no operations are allowed to be chained for demonstrating the primitiveness of our approach. Block RAM-based multiple independent memory banks are programmed as single port, allowing only one read or one write operation during one clock cycle.

```
for ($i = 1; i \le N_i; i++$)
    for ($j = 1; j < N_j; j++$)
    {
        $temp = ((x[i][j-1] * w[i][j+1]) + w[i][j]) * x[i-1][j]$;
        $y[i][j+1] = temp$;
        $temp1 = ((w[i-1][j-1] + temp2) * w[i][j]) + x[i][j+1]$;
        $v[i][j] = temp1$;
        $temp2 = temp - temp1$;
}
```

FIGURE 1. Pseudocode of motivation example.

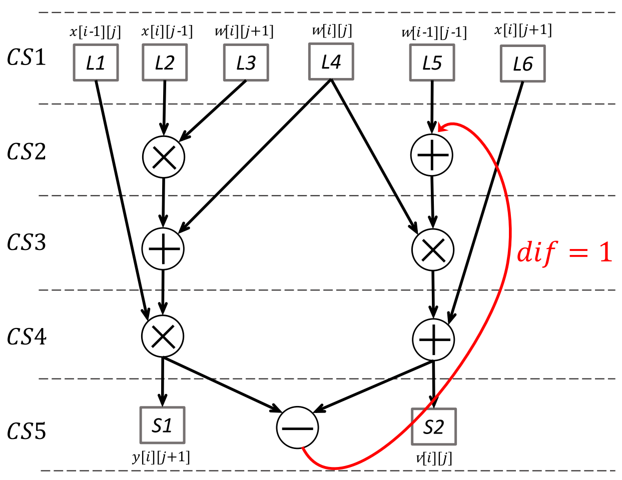


FIGURE 2. Scheduling result of SDC scheduling.

In modulo scheduling, the loop latency (total number of clock cycles required to execute all loop iterations) is $D + (TC - 1) \times II$, where D is single loop latency and TC is the trip count of iterations, which is typically constant for a given loop. Therefore, II is the dominant factor for pipelining performance and minimizing II can significantly improve pipelining performance. The modulo scheduling algorithm begins by calculating the lower bound of II by both the resource constrained *resMII* and the recurrence constrained *recMII*, which is termed Minimum Initiation Interval ($MII = max(\ resMII, recMII)$) [17]. In FPGA-based HLS, resources for operations like adders are typically unconstrained and *resMII* is derived from the constraints of functional units, which are specific in types and number. In addition to calculating *resMII* to obtain a lower bound on II, we also calculate the *recMII*, which indicates the effect of recurrences on II. The recurrences refer to those loop-carried dependencies in DFG. The *recMII* is calculated by dividing the maximum recurrence cycle length by the number of iterations the loop-carried dependency spans (denoted as *dif* in DFG).

In this section, an example is given in Fig.1 to illustrate our joint approach consisting of memory-aware force directed scheduling and multi-cycle memory partitioning algorithms. An iteration of the loop body in Fig.1 is represented as a DFG. First, SDC modulo scheduling gives a legitimate loop pipelining of the DFG, shown in Fig.2. Due to modulo

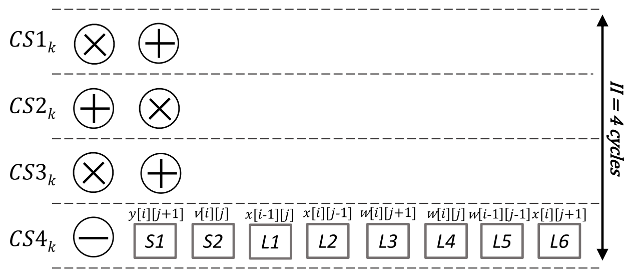


FIGURE 3. Pipelining Kernel of SDC scheduling.

scheduling, the pipelining kernel is achieved by folding the result in Fig.2 by II and the corresponding pipelining kernel is shown in Fig.3. There are eight memory operations simultaneously accessing the memory bank and conflicts exist among these load/store operations in $CS4_k$. The optimal initiation interval (MII) in this example equals 4, while the achieved II in SDC modulo scheduling rises to 11, which is due to the pipelining stalls and is intolerant compared with the single loop latency of only 5 control steps.

Since neither memory access conflict nor memory partitioning is considered in SDC scheduling, memory partitioning technique is additionally supplemented for constructing a memory access conflict-free pipelining kernel. LTB is first applied and the resulting II equals 4 with a cost of 8 banks. The detailed bank mapping after LTB memory partitioning is shown in Fig.6(a).

However, SDC modulo scheduling neglects the interaction between modulo scheduling and memory partitioning, which can cause unnecessary memory access conflicts or waste of memory banks. Then, our proposed joint approach based on memory-aware force directed scheduling (MAFDS+MMP) is applied for comparison. The scheduling result of the MAFDS is shown in Fig.4 and the corresponding pipelining kernel and the optimized memory access pattern are shown in Fig.5. $x[i][j]$ and $w[i][j]$ can be partitioned into only two banks respectively and stored in the way shown in Fig.6(b). The achieved pipelining kernel can execute without any pipelining stall and II turns out to be MII, while the required number of multiple independent memory banks reduces to only 2. It is the optimal result under the constraint of 8 memory access operations and 4 control steps in pipelining kernel.

IV. PROBLEM STATEMENT

A. PRELIMINARIES

In modulo scheduling, the pipelining kernel is achieved by folding the DFG by II. Therefore, the height of a multi-cycle pipelining kernel is defined as II, which corresponds to II control steps or clock cycles. The following terminology is needed to clarify the statement of the problem.

Definition 1 (Memory Access Domain): Given a finite n-dimensional array A, a memory access $\vec{m} \in M$ from array A can be represented as $\vec{m}^A = (m_0^A, m_1^A, \cdots, m_{n-1}^A)^T$, where

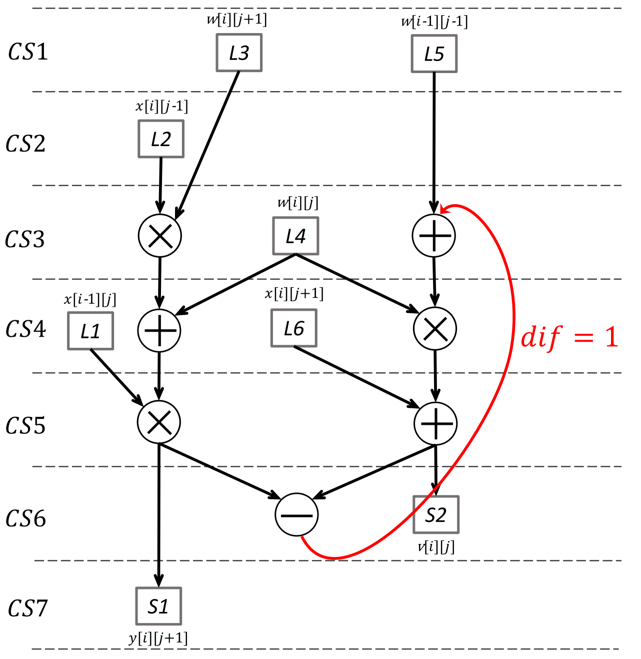


FIGURE 4. Scheduling result of MAFDS scheduling.

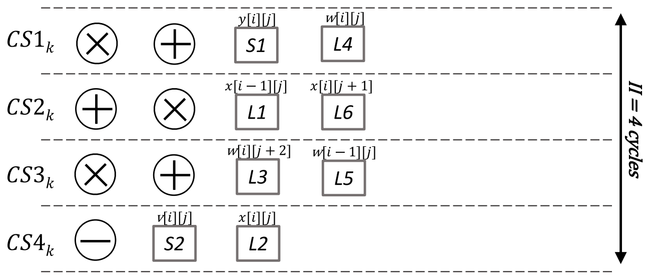


FIGURE 5. Pipelining Kernel of MAFDS scheduling.

$m_i^A \in [0, w_i^A - 1], 0 \le i \le n-1$, and w_i^A indicates the bound of i_{th} dimension in array A.

All the memory accesses from array A in the cs_{th} control step of the pipelining kernel will form a single-cycle access pattern P_{cs}^A, which is defined as follows:

Definition 2 (Single-Cycle Access Pattern): Given the cs_{th} control step (single-cycle), a single-cycle memory access pattern for array A is defined as $P_{cs}^A = \{\vec{\Delta}_{cs}^{A,(1)}, \vec{\Delta}_{cs}^{A,(2)}, \cdots, \vec{\Delta}_{cs}^{A,(r_{cs})}\}$, where r_{cs} indicates the number of memory access operations in the cs_{th} control step. $\vec{\Delta}_{cs}^{A,(k)} = (\Delta_{cs,0}^{A,(k)}, \Delta_{cs,1}^{A,(k)}, \cdots, \Delta_{cs,n-1}^{A,(k)})^T$, $1 \le k \le r_{cs}$, which is a memory access from a finite n-dimensional array A.

In a multi-cycle pipelining kernel, the single-cycle access patterns from different control steps P_{cs}^A correspond to the disparate memory access requirements at different control steps. Therefore, combining with all the single-cycle access patterns in the pipelining kernel, the multi-cycle access pattern P_{kernel}^A is defined.

Definition 3 (Multi-Cycle Access Pattern): Given a pipelining kernel with II control steps, the multi-cycle

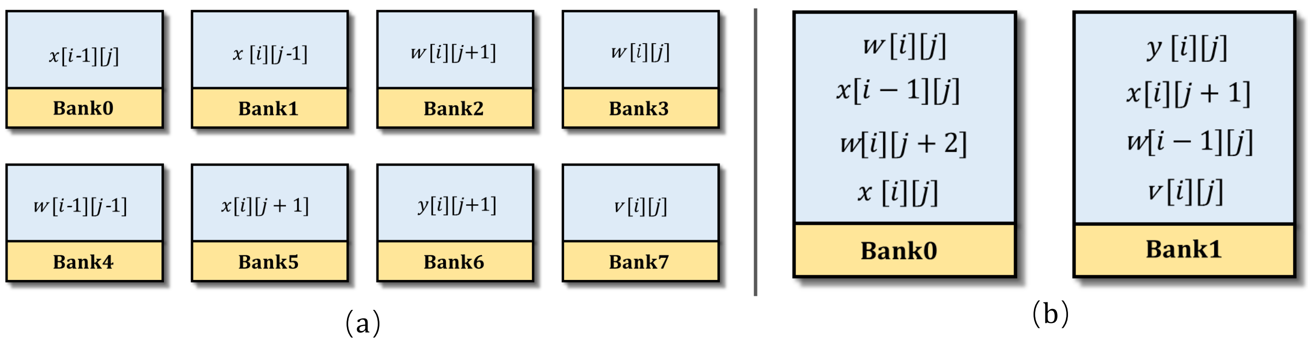


FIGURE 6. Bank mapping result for (a) SDC+LTB. (b) Our Approach (MAFDS+MMP).

access pattern P^A_{kernel} can be represented as $P^A_{kernel} = \{P^A_1, P^A_2, \cdots, P^A_{II}\}$, where $P^A_{cs} = \{\vec{\Delta}^{A,(1)}_{cs}, \vec{\Delta}^{A,(2)}_{cs}, \cdots, \vec{\Delta}^{A,(r_{cs})}_{cs}\}$, $1 \le cs \le II$, indicates the single-cycle access pattern of the pipelining kernel in each control step.

Definition 4 (Valid Bank Mapping): Given arrays $\{A_j | 1 \le j \le q\}$ and the corresponding multi-cycle access patterns $\{P^{A_j}_{kernel} | 1 \le j \le q\}$, a valid bank mapping is described as $(I(\vec{m}), F(\vec{m}))$, where $I(\vec{m})$ represents the bank index that $\vec{m}$ is mapped to and $F(\vec{m})$ represents the inside offset within bank $I(\vec{m})$. The conditions of a valid bank mapping $(I(\vec{m}), F(\vec{m}))$ is presented as follows:

1) $\forall a_1, a_2 \in \{A_1, A_2, \cdots, A_q\}$, where $a_1 \ne a_2$, the corresponding memory accesses $\vec{m}_1$ and $\vec{m}_2$ have unique addresses, i.e. $I(\vec{m}_1) \ne I(\vec{m}_2)$ or $I(\vec{m}_1) = I(\vec{m}_2)$, $F(\vec{m}_1) \ne F(\vec{m}_2)$;
2) $\forall \vec{m}_1, \vec{m}_2 \in \{P^{A_1}_{cs}, P^{A_2}_{cs}, \cdots, P^{A_q}_{cs}\}$, $1 \le cs \le II$, $1 \le j \le q$, we have $\vec{m}_1 \ne \vec{m}_2 \Rightarrow I(\vec{m}_1) \ne I(\vec{m}_2)$, which implies that no memory access conflict exists among the single-cycle access patterns for all the q arrays in any control step.

Definition 5 (Legitimate Pipelining Kernel): Given a DFG $G_d = (V_d, E_d)$, a legitimate modulo scheduled pipelining kernel K_p should satisfy the following conditions:

1) achieved $II \ge MII$;
2) every operation in the DFG is assigned to a particular control step and only appear once in the pipelining kernel;
3) the resulting pipelining kernel does not violate any dependency in the original loop.

B. PROBLEM FORMULATION
Now we can establish the scheduling problem we seek to solve as follows:

Given a DFG $G_d = (V_d, E_d)$ and a set of scheduling constraints C, including resource constraints C_{Res} and recurrence constraints C_{Rec}, find a memory access pattern in the pipelining kernel and a bank mapping such that:

Minimize total number of multiple independent memory banks N_{total}.

Subject to 1) A legitimate pipelining kernel K_p; 2) A valid bank mapping $(I(\vec{m}), F(\vec{m}))$; 3) Minimized II.

V. JOINT APPROACH
A. MMP ALGORITHM
We extend the constructive efficient memory partitioning algorithm in [11] to a multi-cycle context by considering the interaction with the modulo scheduling algorithm. Our proposed multi-cycle memory partitioning (MMP) algorithm is amenable to pipelining kernel with multiple control steps and can ensure a conflict-free memory access pattern, in which all the load/store operations in the same control step can execute without any bank-conflict. The proposed algorithm proceeds in two stages, which are array separation and bank sharing.

1) ARRAY SEPARATION
The first stage of the proposed multi-cycle memory partitioning is array separation. In this stage, the accessed data elements of an array in the pipelining kernel are separated into multiple independent memory banks, guaranteeing that no memory access conflict will occur during any control step. Given the multi-cycle access pattern $P^{A_j}_{kernel}$, $1 \le j \le q$ and II, the minimum number of banks $N^{A_j}_f$ for array A_j can be derived by array separation. The pseudocode for array separation algorithm is presented in **Algorithm 1**.

Since there are II control steps in the pipelining kernel, the union set of all the single-cycle access patterns $\{P^{A_j}_1, P^{A_j}_2, \cdots, P^{A_j}_{II}\}$ will form the multi-cycle memory access pattern $P^{A_j}_{kernel}$ for array A_j (line 2). According the previous definition of multi-cycle access pattern, the single-cycle access pattern can be represented as $P^{A_j}_{cs} = \{\vec{\Delta}^{A_j,(1)}_{cs}, \vec{\Delta}^{A_j,(2)}_{cs}, \cdots, \vec{\Delta}^{A_j,(r_{cs})}_{cs}\}$, $1 \le cs \le II$, and $\vec{\Delta}^{A_j,(k)}_{cs} = (\Delta^{A_j,(k)}_{cs,0}, \Delta^{A_j,(k)}_{cs,1}, \cdots, \Delta^{A_j,(k)}_{cs,n-1})^T$, $1 \le k \le r_{cs}$ (line 4).

Our proposed multi-cycle memory partitioning leverages linear transformation, a fast and low-complexity method, for separating the data elements in an array into different independent memory banks. The maximum differences among the access patterns are calculated respectively (line 5-7). Based on the maximum differences $D^{A_j}_i$, $0 \le i \le n-1$,

Algorithm 1 ArraySeparation

```
Input $\{P_{kernel}^{A_j} | 1 \leq j \leq q\}$, Input $II$,
Output $\{N_f^{A_j} | 1 \leq j \leq q\}$.
1  for $j = 1; j \leq q; j++$ do
2      $\{P_1^{A_j}, P_2^{A_j}, \cdots, P_{II}^{A_j}\} \leftarrow P_{kernel}^{A_j}$;
3      for $cs = 1; cs \leq II; cs++$ do
4          $\{\vec{\Delta}_{cs}^{A_j,(1)}, \vec{\Delta}_{cs}^{A_j,(2)}, \cdots, \vec{\Delta}_{cs}^{A_j,(r_{cs})}\} \leftarrow P_{cs}^{A_j}$;
5          for $i = 0; i \leq n-1; i++$ do
6              $D_{cs,i}^{A_j} \leftarrow$
               $\max_{1 \leq k \leq r_{cs}} \Delta_{cs,i}^{A_j,(k)} - \min_{1 \leq k \leq r_{cs}} \Delta_{cs,i}^{A_j,(k)} + 1$;
7              $D_i^{A_j} \leftarrow \max_{1 \leq cs \leq II} D_{cs,i}^{A_j}$;
8          for $i = 0; i \leq n-2; i++$ do
9              $\alpha_i \leftarrow \Pi_{t=i+1}^{n-1} D_t$;
10         $\alpha_{n-1} \leftarrow \Pi_{t=n}^{n-1} D_t = 1$;
11         $\vec{\alpha}_j \leftarrow \{\alpha_0, \alpha_1, \cdots, \alpha_{n-1}\}$;
12     for $cs = 1; cs \leq II; cs++$ do
13         for $i = 1; i \leq r_{cs}; i++$ do
14             $Z_{cs}^{A_j,(i)} \leftarrow \vec{\alpha}_j \cdot \vec{\Delta}_{cs}^{A_j,(i)}$;
15         $G_{cs} \leftarrow \emptyset$;
16         for $i = 1; i < r_{cs}; i++$ do
17             for $k = i+1; k \leq r_{cs}; k++$ do
18                 $G_{cs} \leftarrow \{G_{cs}, | Z_{cs}^{A_j,(i)} - Z_{cs}^{A_j,(k)} |\}$;
19     $G \leftarrow \bigcup_{1 \leq cs \leq II} G_{cs}$;
20     $N_f^{A_j} \leftarrow \max_{1 \leq cs \leq II} r_{cs}$; // Initialize $N_f^{A_j}$
21     $z \leftarrow 1$;
22     while $zN_f^{A_j} \leq \max G$ do
23         if $zN_f^{A_j} \in G$ then
24             $N_f^{A_j} \leftarrow N_f^{A_j} + 1$;
25             $z \leftarrow 1$;
26         else
27             $z \leftarrow z + 1$;
28 return $\{N_f^{A_j} | 1 \leq j \leq q\}$;
```

the linear transformation vector $\vec{\alpha}_j = (\alpha_0, \alpha_1, \cdots, \alpha_{n-1})$ for array A_j is presented in line 8-11.

After calculating the n-dimensional memory access vector $\vec{m} = (m_0, m_1, \cdots, m_{n-1})^T$, a linear combination of $\vec{m}$ is calculated by $\vec{\alpha} \cdot \vec{m}$. Then, the linear transformation results for the memory access pattern of array A_j in cs_{th} pipelining kernel control step are defined as $(Z_{cs}^{A_j,(1)}, Z_{cs}^{A_j,(2)}, \cdots, Z_{cs}^{A_j,(r_{cs})}) = (\vec{\alpha}_j \cdot \vec{\Delta}_{cs}^{A_j,(1)}, \vec{\alpha}_j \cdot \vec{\Delta}_{cs}^{A_j,(2)}, \cdots, \vec{\alpha}_j \cdot \vec{\Delta}_{cs}^{A_j,(r_{cs})}), 1 \leq cs \leq II, 1 \leq j \leq q$ (line 12-14). For the multi-cycle access pattern, we use set G_{cs} to store the gaps among the linear transformation results $Z_{cs}^{A_j,(k)}$, $1 \leq k \leq r_{cs}$. Precisely, all the absolute values from subtractions between $Z_{cs}^{A,(i)}$ and $Z_{cs}^{A,(j)}$, $1 \leq i < j \leq r_{cs}$, are calculated and stored in G_{cs} and G is a union set of G_{cs}, $1 \leq cs \leq II$ (line 15-19).

Then, $N_f^{A_j}$ is defined to represent the number of banks after array separation for A_j and is initialized by max r_{cs} (line 20). A valid array separation with the current $N_f^{A_j}$ banks should satisfy that all the integral multiples of $N_f^{A_j}$ ($zN_f^{A_j}, z \in \mathbb{Z}^+$) are not in the gap set G. If it fails, the number of banks will increase and the algorithm is repeated until a valid array separation with the given $N_f^{A_j}$ memory banks has been found (line 21-28).

Then, the bank index $I(\vec{m})$ after array separation can be derived by applying cyclic partitioning:

$$I(\vec{m}) = (\vec{\alpha} \cdot \vec{m}) \% N_f^{A_j} \tag{1}$$

After array separation for any given array A_j in the cs_{th} control step, $I_{cs}^{A_j,(i)} = Z_{cs}^{A_j,(i)} \% N_f^{A_j}$ is different from each other, which means all r_{cs} data elements from the same array are separated into different memory banks successfully.

2) BANK SHARING

The next stage of our proposed multi-cycle memory partitioning is bank sharing. Given the multi-cycle memory access patterns $P_{kernel}^{A_j}$, and the corresponding memory bank number $N_f^{A_j}$ after array separation for all the arrays accessed in the pipelining kernel, we can exploit bank sharing algorithm to put the arrays into the same banks according to the memory access conflict states to further reduce the memory bank consumption and derive the minimum number of multiple independent memory banks N_{total} for our proposed joint approach.

Detailed description of bank sharing algorithm is presented in **Algorithm 2**. First, given the multi-cycle memory access patterns $P_{kernel}^{A_j}$, conflict states among the q arrays are extracted and used to determine the $q \times q$ conflict matrix C (line 1). Then, the total number of memory banks N_{total} is initialized by the maximum number of the memory banks $N_f^{A_j}$ after array separation for all the arrays accessed in the pipelining kernel (line 2). According to the array separation results achieved in **Algorithm 1**, the data storage for each accessed array is calculated and stored in a $q \times N_{total}$ matrix S (line 3). Then, the conditions for valid bank sharing are as follows: 1) no conflict exists among the arrays placed in the same bank, and 2) the total storage space for all the arrays in a bank is no bigger than the maximum size of a single bank. Each data storage result in S is checked to determine if they can share the same memory banks (line 5-17). At last, this process will return the total number of memory banks N_{total}, which is the minimum number of memory banks after multi-cycle memory partitioning with the given pipelining kernel and II (line 18).

3) DETAILED PARTITIONING EXAMPLE

In this section, we present the exact steps of our proposed ArraySeparation and BankSharing for the pipelining kernel described in Fig.5.

Algorithm 2 BankSharing

```
   Input $\{P^{A_j}_{kernel} | 1 \le j \le q\}$, Input $\{N^{A_j}_f | 1 \le j \le q\}$,
   Output $N_{total}$.
1  $C_{q\times q} \leftarrow$ GetConflictMat($\{P^{A_j}_{kernel} | 1 \le j \le q\}$);
2  $N_{total} \leftarrow \max_{1\le j\le q} N^{A_j}_f$;
3  $S_l[1, \cdots, N_{total}] \leftarrow S_M$; //Initialize storage space for
   each bank
4  $S_{q\times N_{total}} \leftarrow$ CalDataStorage($\{P^{A_j}_{kernel} | 1 \le j \le q, N_{total}\}$);
5  for $j = 1; j \le q; j++$ do
6      for $k = 1; k \le N_{total}; k++$ do
7          if $S[j][k] > 0$ then
8              $m \leftarrow k$;
               Update$S_l$:
9              if $m > N_{total}$ then
10                 $S_l[m] \leftarrow S_M$;
11                 $N_{total} \leftarrow N_{total} + 1$;
12             if $S_l[m] \ge S[j][k]$&$ConflictFlag == 0$ then
13                 $S_l[m] \leftarrow S_l[m] - S[j][k]$;
14             else
15                 $m \leftarrow N_{total} + 1$;
16                 goto Update$S_l$;
17 return $N_{total}$;
```

First, array $w[i][j]$ and its corresponding access pattern are extracted from the pipelining kernel in Fig.5. Given $II = 4$, the multi-cycle access pattern of $w[i][j]$ are $P^w_{kernel} = \{\{(1,1)^T\}, \emptyset, \{(1,3)^T, (0,1)^T\}, \emptyset\}$ (line 2 in **Algorithm 1**). Then, we have $D^w_0 = max_{1\le cs\le 4} D^w_{cs,0} = max\{1,2\} = 2$ and $D^w_1 = max_{1\le cs\le 4} D^w_{cs,1} = max\{1,3\} = 3$ (line 3-7). Each component of the linear transformation vector $\vec{\alpha}$ is calculated from D^w_0 and D^w_1. Specifically, $\alpha_0 = D_1 = 3$, $\alpha_1 = 1$ and $\vec{\alpha} = (3,1)$ (line 8-11). Then linear transformation results are calculated as follows. $Z^{w,(1)}_1 = (3,1)\cdot(1,1)^T = 4$, $Z^{w,(1)}_3 = (3,1)\cdot(1,3)^T = 6$, and $Z^{w,(2)}_3 = (3,1)\cdot(0,1)^T = 1$ (line 12-14). The corresponding gap set $G_1 = \emptyset$, $G_3 = \{5\}$ and $G = G_1 \bigcup G_3 = \{5\}$ (line 16 -19). Finally, according to the rule described in line 20-28, we can get the resulting minimized bank number $N^w_f = 2$, which means, with our proposed ArraySeparation in **Algorithm 1**, two memory banks can support the parallel memory accesses for array $w[i][j]$ in the whole access pattern. Then, we can continue to perform ArraySeparation on array $x[i][j]$, $y[i][j]$ and $v[i][j]$, respectively. The resulting minimized bank numbers are $N^x_f = 2$, $N^y_f = 1$ and $N^v_f = 1$, respectively.

Without the process of BankSharing proposed in **Algorithm 2**, a scheme of 6 memory banks ($N^w_f + N^x_f + N^y_f + N^v_f = 2+2+1+1 = 6$), as described in Fig.7(a), can guarantee that all the arrays can be access without conflict in any control step of the pipelining kernel. However, we can continue with our BankSharing process to merge the arrays into the same banks

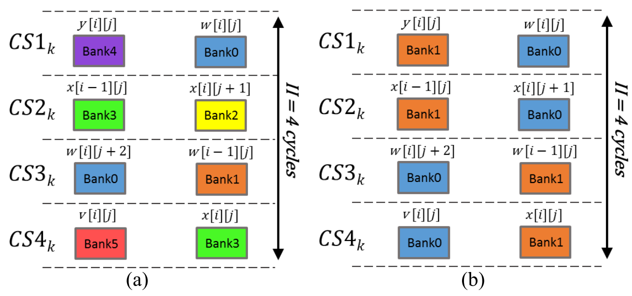


FIGURE 7. (a) An ArraySeparation scheme of 6 memory banks without BankSharing. (b) An ArraySeparation scheme of 2 memory banks after BankSharing.

according to memory access conflict states and the maximum size of a single bank. Finally, after BankSharing, we can achieve a scheme of only 2 memory banks, as described in Fig.7(b), which is a locally conflict-free memory access pattern.

B. MAFDS ALGORITHM

1) MAFDS OVERVIEW

From the insights gained from the aforementioned MMP for the multi-cycle memory access pattern, we introduce the memory-aware force directed scheduling (MAFDS) algorithm to interact with the memory partitioning algorithm by optimizing memory access pattern for reducing the parallel data access demands.

According to the traditional force-directed scheduling [15], different types of the operations in each cycle of the kernel will generate the force for HLS hardware resources. With the definition of multi-cycle access pattern, different control step assignments for the load/store operations in the pipelining kernel correspond to different memory access patterns. In addition, the memory access operations in each control step of the kernel generate the force for multiple independent memory banks. Combined with these features and based on the force-directed scheduling [15], memory-aware force is proposed in MAFDS, which includes the forces for both multi-bank memory and hardware resources.

In HLS, different types of operations require distinct hardware resources. In the proposed MAFDS algorithm, arithmetic and logic operations (e.g. addition, subtraction, multiplication in Fig.4) will first be scheduled separately and only the force for hardware resources $F^R(CS)$ is considered in this step. Lastly, memory access operations are scheduled by calculating the memory-aware force because the preceding schedules of operations will impact the memory-aware force distribution of the memory access operations. Since modulo scheduling has been used and the critical path is extended in MAFDS, the longest distance between two operations in the pipelining kernel is limited to II. Therefore, the mobility of all the memory access operations is defined as II. The main method is presented in **Algorithm 3**. First, mobility are calculated with the given II (line 2). Then, distribution graph $DG(cs_1)$ and changes of probability $v(cs_0, cs_1)$ are calculated

Algorithm 3 Memory-Aware Force Directed Scheduling

```
Input $V_M$, Input $II$, Input $G_d = (V_d, E_d)$,
Output $G_{d\_Sch} = (V_{d\_Sch}, E_{d\_Sch})$.
1 while $V_M \neq \emptyset$ do
2     $MOB \leftarrow$ CalMob($V_M, II, G_d, G_{d\_Sch}$);
3     $(DG, v) \leftarrow$ GetDG($V_M, II, G_d, G_{d\_Sch}$);
4     $Candidates \leftarrow$ Cal$F_B$ ($V_M, DG, v, MOB$);
      //Candidates schedules are selected
5     $(Sch_{LF}, v_{LF}) \leftarrow$
      Cal$F_R$($V_M, DG, v, MOB, Candidates$);
      // A schedule with the lowest force is selected
6     UpdateDFG($Sch_{LF}, v_{LF}, G_d, G_{d\_Sch}$);
      // The selected memory access operation $v_{LF}$ is
      assigned to the control step
7     $V_M \leftarrow V_M \backslash v_{LF}$;
8 return $G_{d\_Sch}$;
```

(line 3), where $DG(cs_1)$ can be represented as a series of springs that will exert forces on operations at control step cs_1, and $v(cs_0, cs_1)$ indicates the increase (or decrease) of the probability on the operation at control step cs_1 due to the operation assignment. The total force $F_v(cs_0)$ associated with the assignment of memory access operation $v \in V_M$ to control step cs_0 is given by $F_v(cs_0) = \sum_{cs_1} DG(cs_1) * v(cs_0, cs_1)$.

Then, the proposed memory-aware force (including $F^B(CS)$ and $F^R(CS)$) is calculated successively and iteratively (line 4-5). The first force $F^B(CS)$ is for memory banks, generated by memory access operations, and the second force $F^R(CS)$ is for hardware resources, generated by the estimated hardware resource consumption for storing the pipelined data in memory access operations in DFG. The candidate schedules are selected by choosing the minimum $F^B(CS)$ forces, which indicates the forces for banks. Then, the lowest $F^R(CS)$ force within the previously selected candidate schedules and the corresponding assignment are finally determined. Next, the scheduled DFG intermediate is updated and the MAFDS algorithm will start over until all the memory access operations are assigned to a control step.

2) DETAILED SCHEDULING EXAMPLE

To show clearly how the memory-aware force directed scheduling works, a detailed example is given in Fig.8, which is an intermediate during MAFDS modulo scheduling after the operations L_1, L_2 and L_4 are scheduled.

First, II is initialized by $MII = 4$ and the mobility of all the memory access operations in Fig.8 is calculated as II, shown in Fig.9. Then, the distribution graphs for both banks and hardware resources are presented in Fig.10 by summing up the force distribution. The overall force generated by one memory access operation is defined as 1. The DG for banks are calculated as follows: the scheduled operations L_1, L_2 and L_4 generate a force distribution of 1 in the corresponding control step separately, while the unscheduled memory

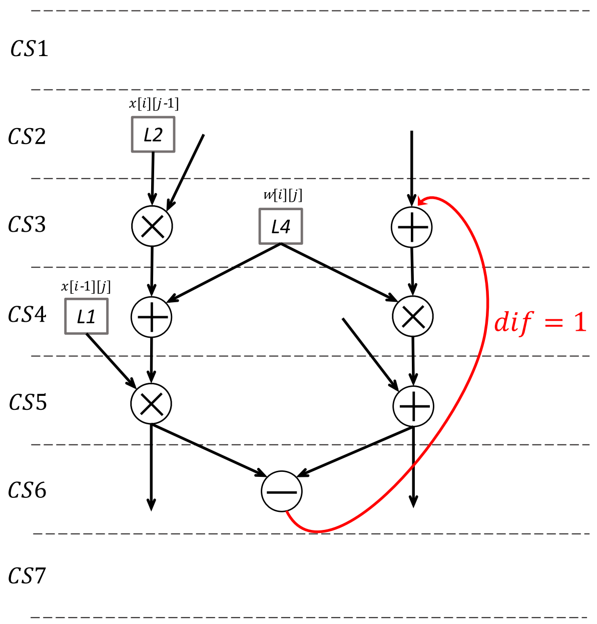


FIGURE 8. An intermediate during MAFDS.

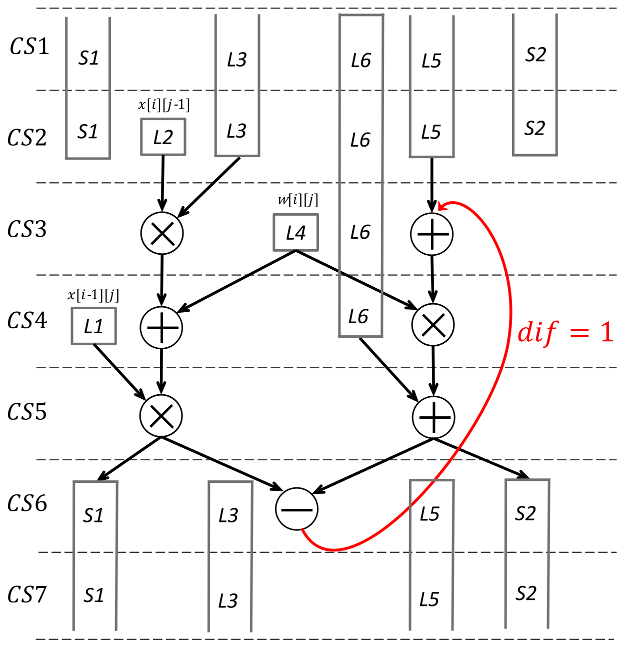


FIGURE 9. Mobility graph of the unscheduled memory access operations in Fig.8. All the memory access operations have a mobility of II = 4.

access operations L_3, L_5, L_6, S_1 and S_2 generate a force distribution for each bank, which is $1/II$ in each control step. Meanwhile, the DG for hardware resources is calculated as follows: besides the scheduled operations L_1, L_2 and L_4, generating a force distribution of 1, operations L_3, L_5, L_6, S_1 and S_2 generate the force distribution for hardware resources, which is non-uniform distribution. Taking L_6 in Fig.9 as an example, its force distribution is $\{1/10, 2/10, 3/10, 4/10\}$ after normalization, based on the estimated register consumption determined by the assignment of memory access operation to different control steps.

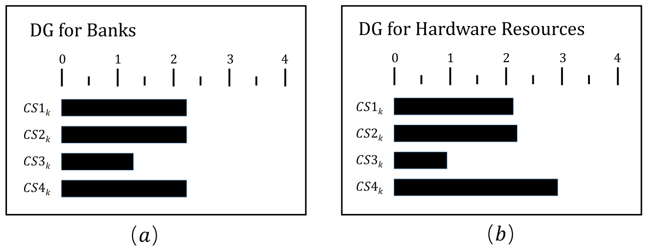


FIGURE 10. **(a) Distribution Graph for Banks. (b) Distribution Graph for Hardware Resources.**

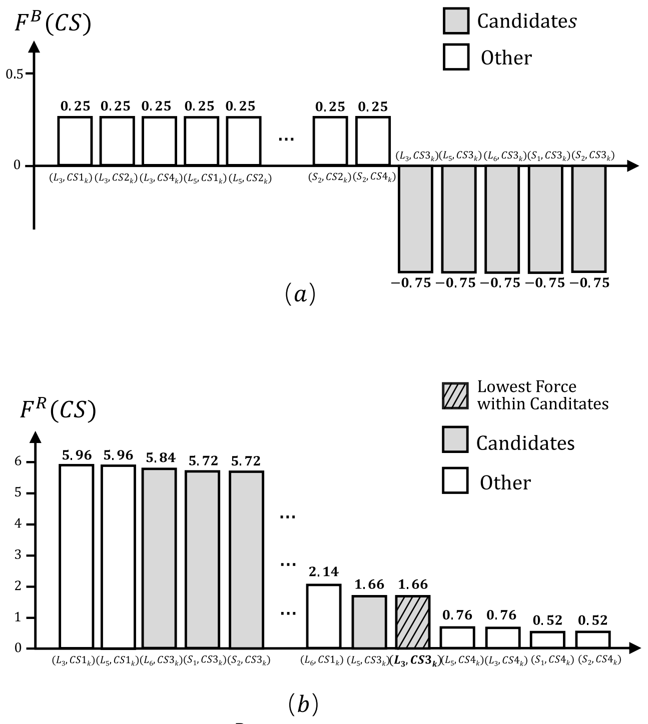


FIGURE 11. **(a) Result of $F^B(CS)$ for the scheduling intermediate in Fig.8. (b) Result of $F^R(CS)$ for the scheduling intermediate in Fig.8.**

With the distribution graph, the memory-aware force are then calculated, shown in a descending order in Fig.11. Several candidate schedules with the minimum $F^B(CS)$ forces are selected in Fig.11(a), which are $(L_3, CS3_k)$, $(L_5, CS3_k)$, $(L_6, CS3_k)$, $(S_1, CS3_k)$ and $(S_2, CS3_k)$. Then in Fig.11(b), the lowest $F^R(CS)$ force within the selected candidates and the corresponding schedule $(L_3, CS3_k)$ are finally determined. This corresponds to assigning the memory access operation L_3 to control step $CS3_k$ in the example intermediate. Our proposed MAFDS is an iterative process until all the memory access operations have been assigned to a control step, presented in **Algorithm 3**.

C. OVERALL FLOW

An overall flow of the proposed joint approach is presented in **Algorithm 4**. First, different types of operations in DFG are classified into several categories and memory access operations are stored into subset V_M (line 1-3). Then, with resource constraints and recurrent constraints, MII is calculated and used to initialize II (line 4-5). With the initialized II,

Algorithm 4 Overall Flow

```
1  $G_d \leftarrow (V_d, E_d)$; // Input DFG
2  $G_{d\_Sch} \leftarrow \emptyset$; // Output Scheduled DFG
3  $V_M \leftarrow \forall v \in V_d$; // Memory access operation subset $V_M$
4  $\{V_{OP1}, V_{OP2}, \cdots\} \leftarrow \forall v' \in V_d \backslash V_M$;
   // Different types of operations (excluding $V_M$) are
   classified into corresponding subsets
5  $II \leftarrow MII$; // Initialize $II$ with Minimum Initiation
   Interval
6  while 1 do
7      for $V_{OP_x} \in \{V_{OP1}, V_{OP2}, \cdots\}$ do
8          FDS($V_{OP_x}, II, G_d, G_{d\_Sch}$);
9      MAFDS($V_M, II, G_d, G_{d\_Sch}$);
10     $\{P^{A_1}_{kernel}, P^{A_2}_{kernel}, \cdots, P^{A_q}_{kernel}\} \leftarrow$
       GetKernel($II, G_{d\_Sch}$);
11     ArraySeparation($\{P^{A_j}_{kernel} | 1 \le j \le q\}, II,$
       $\{N^{A_j}_f | 1 \le j \le q\}$);
12     BankSharing($\{P^{A_j}_{kernel} | 1 \le j \le q\}, \{N^{A_j}_f | 1 \le j \le q\},$
       $N_{total}$);
13     $IsSuccess \leftarrow$ CheckConstraints($C_{Res}, C_{Rec}$);
14     if $IsSuccess$ is true then
15         $Break$;
16     $II \leftarrow II + 1$;
```

the second step is to find an optimized memory access pattern together with a legitimate pipelining kernel which is implemented by memory-aware force directed scheduling algorithm. The operations within $\{V_{OP1}, V_{OP2}, \cdots\}$ will be allocated the control step firstly (line 7-8). Following this, the memory access operations within V_M are scheduled with the proposed memory-aware force directed modulo scheduling (line 9). The third step is to leverage multi-cycle memory partitioning algorithm to achieve an appropriate bank mapping for the multi-cycle pipelining kernel. Once the memory access pattern and pipelining kernel are determined (line 10), the array separation and bank sharing algorithms proposed in **Algorithm 1 and 2** are applied, which can guarantee a conflict-free loop pipelining and derive the minimized total number of multiple independent memory banks N_{total} for the joint approach (line 11-12).

If resource constraints C_{Res} and recurrence constraints C_{Rec} are not satisfied with current II, II will increase by 1 to relax the restriction and a new iteration will start with $II = II + 1$ for a feasible result (line 13-16). This iterative process will stop until the following conditions are satisfied: 1) resource constraints C_{Res} and recurrence constraints C_{Rec} are met, 2) There exists a legitimate pipelining kernel K_p, 3) There is a valid bank mapping $(I(\vec{m}), F(\vec{m}))$.

VI. EXPERIMENTAL RESULTS

The proposed joint approach is implemented based on Legup framework, which is one of the state-of-the-art academic

TABLE 1. Properties of the benchmarks.

BenchMark		Selected Loop	Trip Count	Access#
Polybench	adi	Line 58-78	50*1024*1024	5
	seidel-2d	Line 52-70	20*998*998	9
	jacobi-2d-imper	Line 56-78	10*998*998	5
Livermore	tri-DE	Line 599-615	214000*1000	3
	2-D IHF	Line 1359-1382	172080*5*494	10
	GLRE	Line 1245-1278	623*100	6
Mediabench	dynprog	Line 46-81	18424000	6
	FMID	Line 108-163	8	12
	truespeech	Line 267-316	60*8	3

HLS tools, built on the LLVM compiler [22]. The memory-aware force directed scheduling operates on the LLVM intermediate representation. In multi-cycle memory partitioning, the accessed data elements are mapped to multi-bank memory and the Block RAM-based multiple independent memory banks are programmed as single port for clarifying the proposed algorithms. The typical memory storage capability of each Block RAM is set as 18 Kb. RTL design is then implemented by Xilinx Vivado Design Suite 2015.4 [23] targeting the Virtex-7 VC709 Evaluation Platform (xc7vx690tffg1761-2) for simulation, synthesis and power estimation.

The proposed approach is compared to four other approaches: *Baseline, SDC+LTB, SDC+EMP and SDC+DRMP*. "Baseline" denotes basic SDC modulo scheduling, which works with no memory partitioning. Thus all the banks share an uniform address space and all the accessed arrays are placed continuously. Since there is no other customized approach considering the joint problem of modulo scheduling and memory partitioning in the literature, we combine SDC and three memory partitioning techniques to form different comparative approaches, denoted as "SDC+LTB [9]", "SDC+EMP [11]" and "SDC+DRMP [12]", respectively.

We first evaluate the effect of memory partitioning and modulo scheduling in Section VI-A and VI-B respectively. These experiments verify the usefulness of the proposed multi-cycle memory partitioning (MMP) and memory-aware force directed scheduling (MAFDS) algorithms. Then, the complete experimental results of the joint approach are compared with the other approaches in Section VI-C.

Several representative C loops are selected from three benchmark suites: Polybench, Livermore and Mediabench. Table 1 describes detail, where Trip Count represents the number of loop iterations and Access# represents the number of memory access operations in one loop iteration. In contrast with the previous work, the pipelining performance *II* in our formulation includes not only the interval between starting successive iterations of the loop, but also the pipelining stalls caused by memory access conflicts. Thus, it is more general and precise for evaluating the achieved pipelining performance. Moreover, we also evaluate our approach over the complete *Polybench* suite.

A. COMPARISON OF MEMORY PARTITIONING

The massively parallel data elements accessing the same memory bank simultaneously have made memory access a crucial bottleneck. An urgent request is made for memory partitioning approaches to relieve the memory access problem. In this section, our proposed multi-cycle memory partitioning (MMP) are compared with the state-of-the-art memory partitioning approaches LTB [9], EMP [11] and DRMP [12]. SDC modulo scheduling algorithm is applied as a default scheduling scheme to all of the memory partitioning algorithms for fairness, denoted as SDC+LTB, SDC+EMP, SDC+DRMP and SDC+MMP respectively.

We run the aforementioned C benchmarks and compare the experimental results. Since all the aforementioned memory partitioning approaches can leverage the architecture of multiple independent memory banks to achieve the optimal pipelining performance, no performance benefit in loop pipelining can be observed in this comparison.

We compare MMP to the better one among LTB, EMP and DRMP. As can be seen from the results in Fig.12, the proposed MMP provides an average 21.1% reduction in Bank #, which corresponds to reduction in the number of multiple independent memory banks. This is because the other comparative techniques consider the memory access conflicts from different single-cycle access patterns concurrently, which will result in a globally conflict-free pipelining kernel. While our MMP only targets the single-cycle access pattern in each control step to guarantee a locally conflict-free pipelining kernel. Moreover, the overhead, induced by unmapped interspace segments in Block RAM, can be reduced by 22.9% on average by our proposed MMP. The overhead is measured in the unit of *word*. The overall storage is calculated by the total number of 18 Kb Block RAMs, which has an average reduction of 12.7%.

With the same scheduling algorithm, the generated pipelining datapath and control logic of these memory partitioning algorithms are almost identical, nevertheless, contracted memory subsystems can bring improvements in the resource utilization: Our proposed MMP can reduce the usage of LUT and FF by 12.3% and 11.2% on average. Meanwhile, there is an average reduction of 12.4% in the dynamic power.

B. COMPARISON OF MODULO SCHEDULING

The existing memory partitioning algorithm can guarantee that no memory access conflict will occur in the execution of all the memory access operations in any control step, but state-of-the-art HLS scheduling approaches (e.g. SDC modulo scheduling) ignore the interrelation between modulo scheduling and memory partitioning. Then, our proposed heuristic modulo scheduling algorithm (MAFDS) is compared with SDC modulo scheduling. Likewise, memory

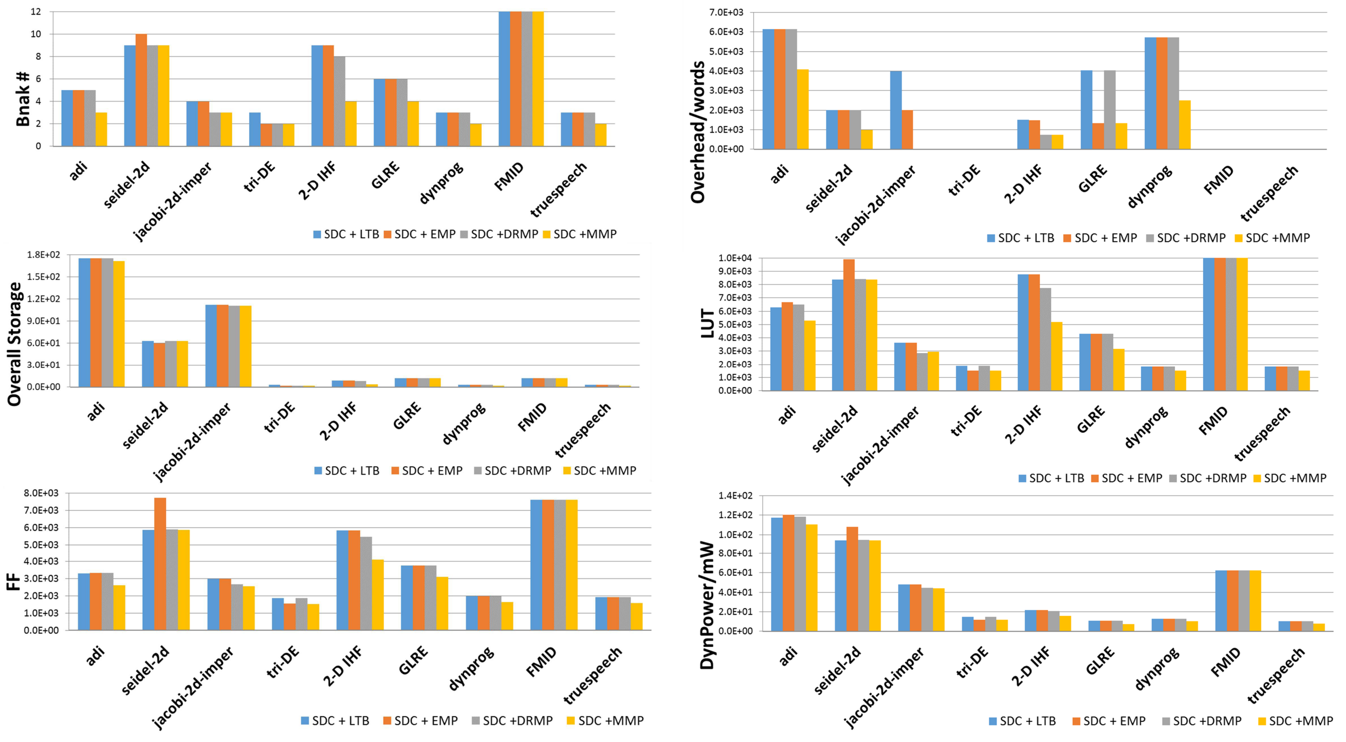


FIGURE 12. Comparison of the experimental results among SDC+LTB, SDC+EMP, SDC+DRMP and SDC+MMP.

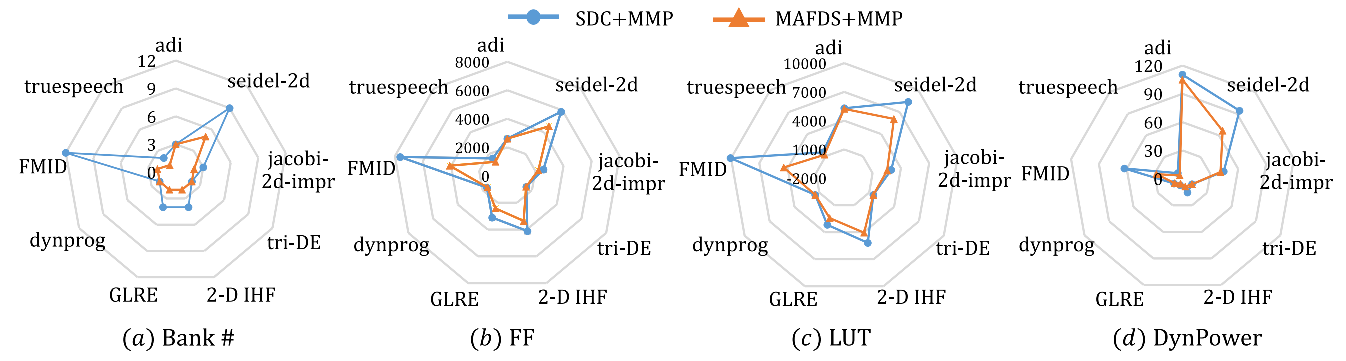


FIGURE 13. Comparison of the experimental results between SDC+MMP and MAFDS+MMP.

partitioning algorithm MMP is applied to both of the modulo scheduling algorithms for fairness, denoted as SDC+MMP and MAFDS+MMP respectively.

The comparison results are presented in Fig.13. Over the representative benchmarks, the number of multiple independent memory banks has an average reduction of 34.6%. This is because our MAFDS can optimize the multi-cycle access pattern to reduce the demand for parallel data accesses (reducing maximum number of load/store operations in any control step) and achieve the optimal multi-bank consumption. The results also show that our scheduling approach can result in an average reduction of 18.5% in LUT and 16.2% in FF, which is due to both the reduction of resources for storing the pipelined data and the contraction in memory subsystem. With MAFDS, the dynamic power is reduced by 23.7% on average versus the comparative work.

C. EXPERIMENTAL RESULTS ON SELECTED LOOPS

The experimental results and comparisons over 9 C loops from three different benchmark suites are presented in Table 2. "MAFDS+MMP" is our proposed joint approach, which consists of multi-cycle memory partitioning (MMP) and memory-aware force directed scheduling (MAFDS) algorithms.

When comparing "Baseline" with "Our Approach", a great improvement is achieved in the pipelining performance in our proposed scenario, with an average improvement of 43.0% for *II* and 42.8% for latency. It is noteworthy

TABLE 2. Experimental results and comparisons on multi-benchmark suites.

Benchmark	Method	II	Latency/cycles	Bank #	Overhead /words	OverallStorage	LUT	FF	DSP	DynPower /mW	ClockPeriod /ns
adi	Baseline	5	262144008	1	0	171	4082	1877	6	75.6	2.195
	SDC + LTB	3	157286405	5	6144	175	6292	3311	6	117.2	2.253
	SDC + EMP	3	157286405	5	6144	175	6666	3327	6	120.4	2.253
	SDC + DRMP	3	157286405	5	6144	175	6485	3320	6	118.5	2.253
	Our Approach	3	157286405	3	0	171	5231	2611	6	105.0	2.195
seidel-2d	Baseline	9	179280734	1	0	55	3473	3002	0	33.4	8.104
	SDC + LTB	2	39840167	9	1996	63	8389	5866	0	94.2	8.104
	SDC + EMP	2	39840167	10	1996	60	9883	7739	0	107.9	8.104
	SDC + DRMP	2	39840167	9	1996	63	8401	5902	0	94.8	8.104
	Our Approach	2	39840168	5	1996	55	6069	4541	0	66.4	8.104
jacobi-2d-imper	Baseline	5	49800208	1	0	109	1942	1843	0	36.2	5.343
	SDC + LTB	3	29880125	4	3992	112	3623	2993	0	47.9	5.689
	SDC + EMP	3	29880125	4	1996	112	3623	2993	0	47.9	5.689
	SDC + DRMP	3	29880125	3	0	111	2873	2669	0	44.3	5.689
	Our Approach	3	29880126	2	0	110	2543	2212	0	41.1	5.396
tri-DE	Baseline	3	642000005	1	0	1	1155	1218	3	9.0	4.906
	SDC + LTB	2	428000004	3	6	3	1915	1880	3	15.0	4.906
	SDC + EMP	2	428000004	2	0	2	1509	1543	3	11.8	4.906
	SDC + DRMP	2	428000004	3	6	3	1915	1880	3	15.0	4.906
	Our Approach	2	428000004	2	0	2	1504	1520	3	11.8	4.906
2-D IHF	Baseline	10	4250376014	1	0	1	3501	3052	15	12.5	8.636
	SDC + LTB	6	2550225610	9	1498	9	8765	5845	15	22.0	8.704
	SDC + EMP	6	2550225610	9	1482	9	8765	5845	15	22.0	8.704
	SDC + DRMP	6	2550225610	8	749	8	7747	5461	15	20.8	8.704
	Our Approach	6	2550225612	2	494	2	4058	3362	15	8.8	8.636
GLRE	Baseline	5	311505	1	0	11	2012	2119	6	3.8	4.906
	SDC + LTB	3	186904	6	4044	12	4319	3761	6	11.1	5.936
	SDC + EMP	3	186904	6	1348	12	4362	3770	6	11.1	5.936
	SDC + DRMP	3	186904	6	4044	12	4319	3761	6	11.1	5.936
	Our Approach	3	186905	2	0	12	2433	2420	6	5.7	4.906
dynprog	Baseline	3	55272005	1	0	1	1185	1323	0	8.7	5.580
	SDC + LTB	2	36848004	3	5724	3	1834	1977	0	13.0	5.838
	SDC + EMP	2	36848004	3	5724	3	1804	1955	0	12.7	5.838
	SDC + DRMP	2	36848004	3	5724	3	1834	1977	0	13.0	5.838
	Our Approach	2	36848004	2	2500	2	1494	1607	0	9.5	5.752
FMID	Baseline	16	147	1	0	1	3724	3906	18	27.7	4.906
	SDC + LTB	6	57	12	0	12	10058	7629	18	62.6	4.906
	SDC + EMP	6	57	12	0	12	10058	7629	18	62.6	4.906
	SDC + DRMP	6	57	12	0	12	10058	7629	18	62.6	4.906
	Our Approach	6	57	2	0	2	4411	4098	18	28.0	4.906
truespeech	Baseline	5	2407	1	0	1	1198	1277	3	4.6	5.969
	SDC + LTB	4	1926	3	0	3	1819	1919	3	10.3	6.335
	SDC + EMP	4	1926	3	0	3	1819	1919	3	10.3	6.335
	SDC + DRMP	4	1926	3	0	3	1819	1919	3	10.3	6.335
	Our Approach	4	1927	1	0	1	1198	1292	3	4.4	6.214

that with our proposed joint approach, hardware resource usage will increase and more power is consumed, but it is still an acceptable cost compared with the considerable improvement in pipelining performance.

For the rest of these comparative approaches, we compare the results of ''Our Approach'' with the best results achieved among ''SDC+LTB'', ''SDC+EMP'' and ''SDC+DRMP''. The results show that our joint solution can further optimize the hardware resource usage, while still achieving the optimal pipelining performance (*II* and latency reach the optimal minimum values). Table 2 shows that the average improvements in Bank # and storage overhead are 49.2% and 32.3% respectively to the best of the other 3 approaches. There is also an average reduction of 30.0% in overall storage (number of 18Kb Block RAMs) after memory partitioning. The contraction of memory subsystem by our approach contributes to the reduction in LUT (28.3%) and FF (25.9%). Furthermore, the total dynamic power consumption is reduced by 32.3% and the clock period is reduced by 3.2% on average.

It is worthy to note that the single loop latency achieved by ''Our Approach'' will be slightly greater than those achieved by the comparative methods (an increase of a few cycles in some benchmarks). This is because our approach will increase the critical path to guarantee that there is no pipelining stall in the loop pipelining and an optimized memory access pattern can be achieved. However, with the trip count being large enough, the tiny increment in loop latency can

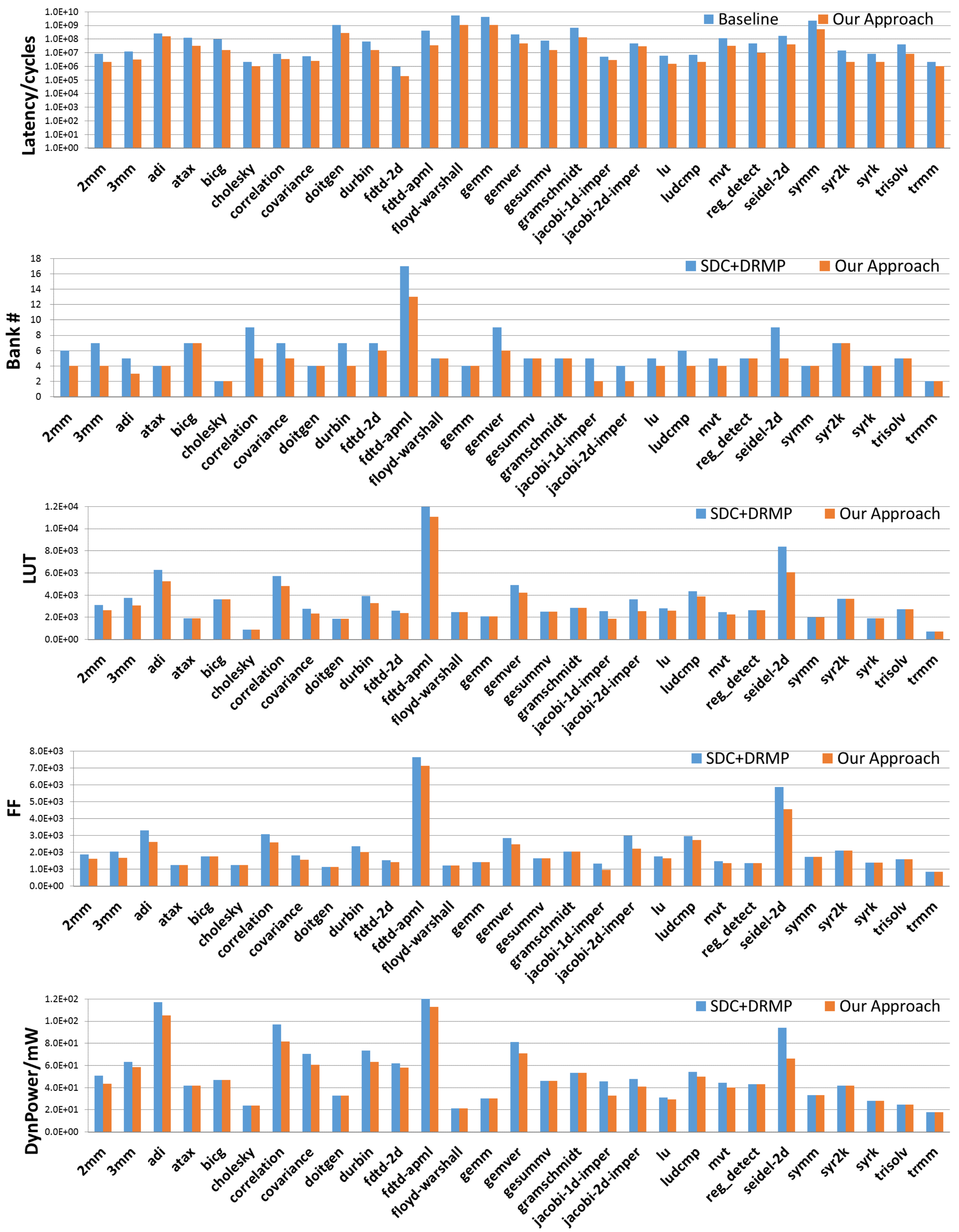


FIGURE 14. Experimental results and comparisons on complete Polybench suite.

always be neglected and no distinct variation will be observed in the corresponding wall-clock time.

The improvement in our joint approach is mainly from two aspects: 1) the memory partitioning in MMP separates the accessed arrays into multiple independent memory banks and the memory access conflicts in the same control step will be avoided. Meanwhile, taking the multi-cycle pipelining kernel achieved by modulo scheduling into collaborative consideration, less banks are required for a conflict-free pipelining kernel. 2) the memory-aware force directed scheduling has been applied to further optimize the multi-cycle access pattern and reduce the parallelism of the load/store operations in each control step of the pipelining kernel and it will interact with MMP to reduce the hardware resource usage and memory bank consumption.

D. EXPERIMENTAL RESULTS ON COMPLETE POLYBENCH SUITE

In this section, we present the experimental results on the complete Polybench suite. Since "SDC+DRMP" achieves relative better performance than "SDC+LTB" and "SDC+EMP" in previous experiments, we compare our approach to both "Baseline" and "SDC+DRMP".

The complete experimental results and comparisons are presented in Fig.14. When comparing "Baseline" with our approach, a great improvement is achieved in the pipelining performance in our proposed joint approach, with an average improvement of 70.4% in total latency.

Both "SDC+DRMP" and our approach achieve the optimal pipelining performance, but the results show that our joint solution can further optimize the hardware resource usage, while still achieving the optimal pipelining performance. Fig.14 shows an average improvement of 18.3% in Bank #. There is also an average reduction in LUT (8.3%) and FF (7.7%) achieved by our approach. Furthermore, the total dynamic power consumption is reduced by 6.7% on average. All these results, which are consistent with the experiments over the selected loops, unanimously prove the effectiveness of our joint approach for improving loop performance in HLS.

VII. CONCLUSION

In this paper, we study the problem of HLS modulo scheduling in the context of multi-bank memory and propose a joint memory-aware force directed scheduling and multi-cycle memory partitioning approach for the massively parallel memory access bottleneck problem. Compared with the state-of-the-art approaches, experimental results demonstrate that the proposed joint approach can produce substantial reduction in the number of multiple independent memory banks, while achieving the optimal loop pipelining performance.

REFERENCES

[1] D. Koch, D. Ziener, and F. Hannig, "FPGA versus software programming: Why, when, and how?" in *FPGAs for Software Programmers*. Cham, Switzerland: Springer, 2016, pp. 1–21.

[2] M. Fingeroff, *High-Level Synthesis Blue Book*. Bloomington, IN, USA: Xlibris, 2010.

[3] A. Morvan, S. Derrien, and P. Quinton, "Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 3, pp. 339–352, Mar. 2013.

[4] W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2013, pp. 9–18.

[5] J. Liu, J. Wickerson, and G. A. Constantinides, "Loop splitting for efficient pipelining in high-level synthesis," in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2016, pp. 72–79.

[6] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan, "Software pipelining," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 367–432, 1995.

[7] J. M. Codina, J. Llosa, and A. González, "A comparative study of modulo scheduling techniques," in *Proc. 16th Int. Conf. Supercomput.*, Jun. 2002, pp. 97–106.

[8] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," *Nature*, vol. 184, no. 4692, p. 1010, 2008.

[9] Y. Wang, P. Li, P. Zhang, C. Zhang, and J. Cong, "Memory partitioning for multidimensional arrays in high-level synthesis," in *Proc. 50th Annu. Design Autom. Conf.*, 2013, p. 12.

[10] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2014, pp. 199–208.

[11] C. Meng, S. Yin, P. Ouyang, L. Liu, and S. Wei, "Efficient memory partitioning for parallel data access in multidimensional arrays," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.

[12] J. Su, F. Yang, X. Zeng, and D. Zhou, "Efficient memory partitioning for parallel data access via data reuse," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2016, pp. 138–147.

[13] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proc. 43rd Annu. Design Autom. Conf.*, 2006, pp. 433–438.

[14] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul./Aug. 2009.

[15] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, no. 6, pp. 661–679, Jun. 1989.

[16] C. Mandal, P. Chakrabarti, and S. Ghose, "Complexity of scheduling in high level synthesis," *VLSi DESiGN*, vol. 7, no. 4, pp. 337–346, 1998.

[17] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Annu. Int. Symp. Microarchitecture*, 1994, pp. 63–74.

[18] J. Llosa, E. Ayguadé, A. Gonzalez, M. Valero, and J. Eckhardt, "Lifetime-sensitive modulo scheduling in a production environment," *IEEE Trans. Comput.*, vol. 50, no. 3, pp. 234–249, Mar. 2001.

[19] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2013, pp. 211–218.

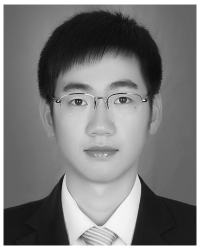
[20] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC scheduling with recurrence minimization in high-level synthesis," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2014, pp. 1–8.

[21] S. Yin, X. Yao, T. Lu, L. Liu, and S. Wei, "Joint loop mapping and data placement for coarse-grained reconfigurable architecture with multi-bank memory," in *Proc. 35th Int. Conf. Comput.-Aided Design*, 2016, p. 127.

[22] R. Nane *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[23] Xilinx. *Xilinx Vivado Design Suite*. Accessed: Dec. 2016. [Online]. Available: http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2015-4.html

SHOUYI YIN received the B.S., M.S., and Ph.D. degrees in electronic engineering from Tsinghua University, China, in 2000, 2002, and 2005, respectively. He was with Imperial College London as a Research Associate. He is currently with the Institute of Microelectronics, Tsinghua University, as an Associate Professor. His research interests include reconfigurable computing, mobile computing and SoC design.



TIANYI LU received the B.S. degree from Southeast University, Nanjing, China, in 2015. He is currently pursuing the M.S. degree with the Institute of Microelectronics, Tsinghua University, Beijing, China. His research interests include reconfigurable computing and optimization of compiler for reconfigurable computing.

XIANQING YAO was born in 1989. He received the B.S. degree from the School of Optoelectronic Engineering, Nanjing University of Posts and Telecommunications, Nanjing, China, in 2012. He is currently pursuing the M.S. degree with the Institute of Microelectronics, Tsinghua University, Beijing, China. His research interests include reconfigurable computing and optimization of compiler for reconfigurable computing.

ZHICONG XIE received the B.S. degree in computer science and technology from Zhejiang University, Zhejiang, China, in 2014. He is currently pursuing the M.S. degree with the Institute of Microelectronics, Tsinghua University, Beijing, China. His current research interests include computer architecture, CAD, and VLSI.

LEIBO LIU received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China, in 1999 and the Ph.D. degree from the Institute of Microelectronics, Tsinghua University, in 2004. He currently serves as an Associate Professor with the Institute of Microelectronics, Tsinghua University. His research interests include reconfigurable computing, mobile computing, and VLSI DSP.

SHAOJUN WEI was born in Beijing, China, in 1958. He received the Ph.D. degree from Faculte Polytechnique de Mons, Belgium, in 1991. He became a Professor with the Institute of Microelectronics, Tsinghua University in 1995. He is currently a Senior Member with the Chinese Institute of Electronics. His main research interests include VLSI SoC design, EDA methodology, and communication ASIC design.