Optimizing Imperfectly-Nested Loop Mapping on CGRAs via Polyhedral-Guided Flattening

Xingyu Mo, Yawen Li, Dajiang Liu,
College of Computer Science, Chongqing University, Chongqing 400044, China
Email: momoxy6@gmail.com, moxiaosi@hotmail.com, liudj@cqu.edu.cn

***Abstract*—Coarse-Grained Reconfigurable Arrays (CGRAs) offer a promising balance between high performance and power efficiency. To reduce the invocation overhead when mapping an imperfectly nested loop, loop flattening is used to transform the nested loop into a single-level loop. However, loop flattening not only leads to a big loop body but also has a narrow application scope. To this end, this work proposes a polyhedral model-based loop flattening approach for imperfectly nested loop mapping. By exploring loop structures via polyhedral transformation, we can find a flattening-friendly loop structure with more data reuse opportunities and reduced sibling loops, resulting in improved loop pipelining performance. Experimental results demonstrate a remarkable (1.37-1.62×) speedup compared to the state-of-the-art approaches while maintaining short compilation times.**

***Index Terms*—CGRA, Loop Flattening, Polyhedral Model**

I. Introduction

With the increased computational demands from emerging applications, it has driven architectures to be both energy-efficient and flexible. While ASICs offer high performance and power efficiency, their lack of programmability leads to high costs and a long time to market. Although FPGAs could provide bit-level configuration flexibility, their fine-grained computation and control contribute to significant power consumption. Coarse-Grained Reconfigurable Array (CGRA) [1] [2] is a promising parallel computing architecture that combines high energy efficiency and flexibility, enabling energy-efficient solutions for a wide range of applications.

A typical CGRA, as shown in Fig. 1(a), consists of a host controller, a data memory, and many ALU-like Processing Elements (PEs) arranged in a 2-D array. Via the Load-Store-Units (LSUs), the Processing Element Array (PEA) can load data from or store data in each bank of the data memory. Via the local register file in PEs and flexible inter-PE connections, on-PEA data reuse [2] could be achieved for short-distance memory accesses. To realize the potential of CGRA, computation-intensive parts of the application, e.g., loops, are mapped to CGRA for acceleration. For single-level loops, modulo scheduling [3] is widely used to pipeline the loop execution by overlapping the execution of Data Flow Graphs (DFGs) extracted from adjacent loop iterations, where the Initiation Interval (II) between iterations is the main concern. However, most loops in real-life applications are imperfectly nested loops involving statements outside the innermost loop

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). (*Corresponding author: Dajiang Liu*)

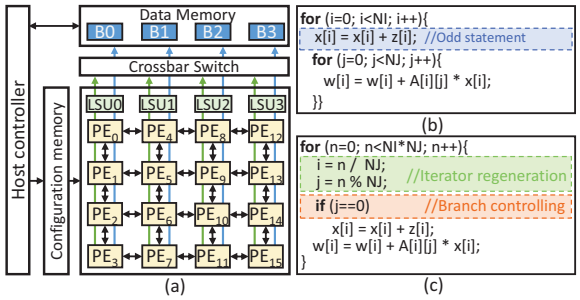

Fig. 1. The target CGRA and an example. (a) The target CGRA supports iterator regeneration, (b) an imperfectly nested loop, and (c) the flattened loop.

(odd statements) or multiple inner loops (sibling loops), as shown in Fig. 1(b), it is a challenging task to pipeline these imperfectly-nested loops.

To address this problem, previous methods can be categorized into two parts: 1) partial loop pipelining [3] and 2) full loop pipelining [4]. Partial loop pipelining only focuses on the pipelining of the innermost or innermost two loops of a nested loop on the CGRA, while the outer loops are executed sequentially on the host controller. To improve pipelining performance by exploring different variants of the innermost loop, a polyhedral-based pipelining approach is proposed in [5]. This approach aims to find a more favorable loop structure through loop transformations. For the pipeline's prologue and epilogue and context switching between different pipelining kernels, these works of partial loop pipelining have significant overheads from outer loops' invocation.

The full loop pipelining attempts to reduce invocation overhead and communications by flattening a nested loop into a single-level loop. However, as shown in Fig. 1(c), loop flattening inevitably introduces some *side affects*, i.e., division and modulo operations for iterator regeneration and branch operations for conditional execution, leading to a larger loop body. For conditional execution on CGRAs, partial prediction [1] is commonly adopted, where all possible branches will be executed for result selection, further leading to a large loop body. To reduce the side effect of iterator regeneration, an extended PE [4] is proposed, simplifying division and modulo operations into counters and comparators. As loop flattening cannot support loops including sibling loops, loop fission is adopted in [4] to split sibling loops into separate loop kernels. However, loop fission is not always applicable when condensing data dependence. Notably, if loop fission fails to eliminate the sibling loops, it has to execute the nested loop in a partial loop pipelining fashion, leading to a significant performance loss.

979-8-3503-4859-0/DATE24/© 2024 EDAA

As discussed above, flattening-based pipelining effectively reduces the invocation overhead. If we can overcome its side effects (reducing loop body) and extend its application scope (eliminating sibling loops), the pipelining performance of imperfectly nested loops could be greatly improved. Considering the potency of polyhedral model [6] in loop transformation, it offers opportunities to find flattening-friendly loop structures for CGRA execution, with more data reuse and fewer sibling loops. To this end, we propose a polyhedral model-based loop flattening approach for imperfectly nested loop mapping. The contributions of this work are summarized as follows:

- We propose a polyhedral-based loop flattening approach that could fully and effectively explore the structure of a given nested loop to find a flattening-friendly loop variant.
- We propose an efficient solution for the above loop transformation problem by introducing sibling loop elimination and innermost-loop evaluation such that an optimized transformation could be found quickly.

II. Background and Related Works

A. The Notations in Polyhedral Model

Definition 1: Data Dependence Graph (DDG) [5]. DDG is a directed multi-graph where each vertex represents a statement and an edge, $e \in E$ (dependence set) from vertex V_i to V_j represents a polyhedral dependence from statement S_i to S_j.

Definition 2: Strongly Connected Component (SCC) [5]. SCC is the strongly connected component in a DDG. All the statements belonging to an SCC cannot be distributed further.

Definition 3: Reuse Vector [7]. If the access of source iteration $\vec{i}$ is a write and the target iteration $\vec{j}$ is a read, the Read-After-Write (RAW) reuse vector is $\vec{d} = \vec{j} - \vec{i}$. Similarly, if the access of source iteration $\vec{i}$ and target iteration $\vec{j}$ are both a read, the Read-After-Read (RAR) reuse vector is $\vec{r} = \vec{j} - \vec{i}$.

Definition 4: Statement-wise Transformation [6]. If S be a statement in the program with a dimension of m_S, the statement-wise transformation of S is represented as follows:

$$\phi_S^k(\vec{i_S}) = \left(c_1^k, c_2^k, \cdots c_{m_S}^k\right)\left(\vec{i_S}\right) + c_0^k, k \in [0, 2m_s] \quad (1)$$

If $k = 2n$ ($n \in [0, m_S]$), $\phi_S^k(\vec{i_S})$ is called splitter indicating the textual order of statements, which is satisfied $(c_1^k, c_2^k, ...c_{m_S}^k) = \vec{0}$. Otherwise, $\phi_S^k(\vec{i_S})$ is called hyperplane or 1-D affine transformation indicating the iteration order of statements which is satisfied $(c_1^k, c_2^k, \cdots c_{m_S}^k) \neq \vec{0} \wedge c_0^k = 0$. By finding the splitters and the 1-D hyperplanes in an interleaved approach, starting from the outermost to the innermost layers, we can determine a full transformation (Φ_S) for a nested loop.

B. Related Works

Over the past decade, the polyhedral model has firmly established itself as a robust intermediate representation for high-level optimizations in compilers. In [8] [9], the authors have been reducing tensor expressions to polyhedral representations and utilizing a polyhedral scheduler to execute extensive transformations, which effectively automates memory management for Neural Processing Units. In [7], polyhedral model is used to represent iteration distances among relevant array instances, maximizing the utilization of external memory bandwidth. In [10], polyhedral transformations are performed to map the operator's iteration space onto the PE array, automatically discovering all possible system array designs, and selecting the optimal design to achieve peak performance. In [5], polyhedral model is formulated for partial loop pipelining on CGRAs, which employs an exhaustive search to find the optimal solution. As the loop structure for flattening on CGRAs involves special constraints [4] (e.g., no sibling loops and regular iteration domain) and a unique objective function (full loop pipelining performance), existing polyhedral models mentioned above cannot be directly adopted. Therefore, we will formulate a new optimization problem based on polyhedral model for loop flattening on CGRAs.

III. Motivation

Fig. 2 presents a motivating example to demonstrate the effectiveness of our approach. As shown in Fig. 2(a), the input is an imperfectly-nested loop including 2 statements (S_1 and S_2) with 1 memory-storing operation (ST_0) and 3 memory-loading operations (L_0, L_1 and L_2), which involves two loop-carried dependencies (blue arrow and red arrow). As depicted in Fig. 2(g), the target CGRA consists of a 2×2 PEA with 2 LSUs, where each PE includes 2 local registers.

Fission-based loop flattening. To flatten this imperfectly nested loop involving sibling loops, the conventional approach in [4] would split it into two perfect loops, and then perform loop flattening on kernel 1 and kernel 2, separately. Fig. 2(b) presents the transformed loop after loop fission and flattening, where iterator generators (IGs) are introduced to regenerate the original iteration variables (i and j). Then, each flattened loop kernel is converted into a DFG, which involves extra operations of address generation for memory access operations (e.g., L_1 and L_2). Fig. 2(c) presents the generated DFG for kernel 2, where dashed nodes indicate the extra operations for iterator generation and address generation while solid nodes indicate the substantial operations in statement S_2. For valid routing, routing nodes (R_1 and R_2) are also inserted in the DFG. Finally, as shown in Fig. 2(d), the generated DFG can be successfully mapped onto the target CGRA with II=3, where nodes in different colors indicate that they are from different iterations. With the addition of kernel 1 in Fig. 2(b), the execution time of this fission-based loop flattening will be even longer.

Comprehensive-transformation-based loop flattening. Fig. 2(e) and (f) demonstrate the loop flattening result after a series of comprehensive loop transformations. Firstly, loop fusion is performed such that the two statements in sibling loops can be merged into the same inner loop. After this, as the iteration domain shown in Fig. 2(h), the RAW reuse of the first dependence (blue arrow) from the storing operation (ST_0) in S_1 to the second loading operation (L_2) in S_2 could be reduced to [0,0]. Then, loop interchange is further performed and the RAW reuse of the second loop-carried dependence from the storing operation ST_0 in S_1 to the first loading operation (L_1) in S_2 could be reduced. As the red arrows depicted in Fig. 2(i), the RAW reuse of the second dependence is reduced from [1,0] to [0,1]. After loop fusion and interchange, as illustrated in Fig.

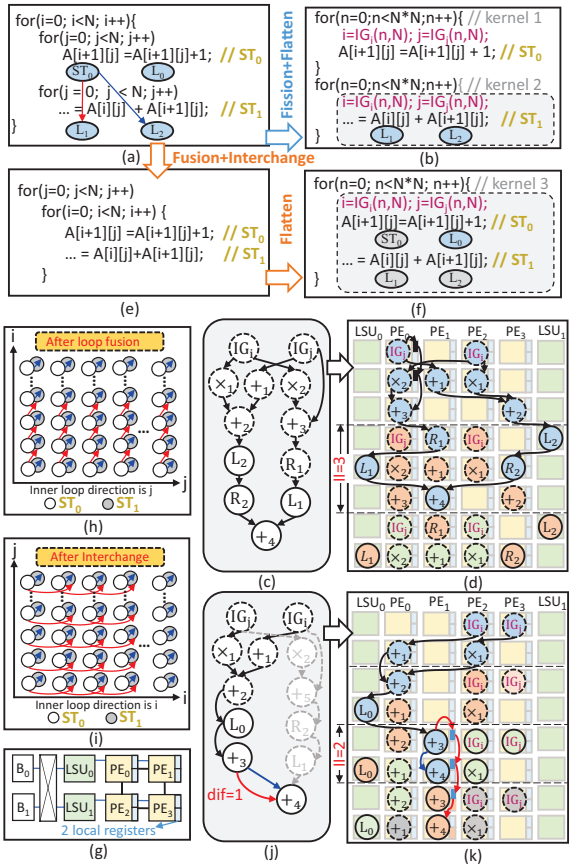

Fig. 2. A motivating example. (a) The original nested loop, (b) the transformed loop after loop fission and loop flattening, (c) the generated DFG of kernel 2 after loop fission and flattening, (d) the mapping result of kernel 2 with II=3, (e) the transformed loop after loop fusion and interchange, (f) the transformed loop (kernel 3) after further loop flattening, (g) the target 2×2 CGRA with two LSUs, (h) the iteration space after loop fusion, (i) the iteration space after loop interchange, (j) the generated DFG of kernel 3, and (k) the mapping result of kernel 3 with II=2.

2(f), we obtain the transformed loop using loop flattening. Since the two dependencies from S_1 to S_2 become intra-iteration dependence and inner-loop carried dependence, respectively, they could be trivially routed via intra-PEA resources. Consequently, memory access nodes (ST_0, L_1 and L_2) and their address generation operations could be eliminated. As shown in Fig. 2(j), the generated DFG only includes 8 operations including an intra-iteration dependence (blue arrow) and an inner-loop-carried dependence (red arrow) with dif=1 (i.e., it traverses 1 iteration in the flattened loop), where the grey nodes indicate the eliminated nodes. Finally, as shown in Fig. 2(k), the generated DFG can be successfully mapped onto the target CGRA with II=2, where the blue dependence and red dependence are routed via the result register and local register in PE_1, respectively. As a result, the execution time of this comprehensive-transformation-based loop flattening is greatly reduced, as compared to the fission-based one.

IV. Problem Formulation

In this section, we formulate the flattening-friendly loop transformation into an optimization problem.

A. Optimization Problem of Flattening-friendly Transformation

Given a nested loop P with Static Control Part (SCoP) [6] and rectangular iteration domain and a target CGRA, the flattening-friendly transformation problem is to find a loop transformation Φ **such that**:

1) The data dependencies in P should be held.
2) The iteration domain of the newly transformed loop (P') remains rectangular for loop flattening [4].
3) The P' does not include sibling loops.
4) The Total Execution Cycles (TEC) of P' on the target CGRA after loop flattening are minimized.

In the above optimization problem, the former two conditions will be satisfied in affine transformations, and the third one will be satisfied in the loop fusion/fission proposal. Since the transformed program only includes one or more single-level kernels that could be fully pipelined, the TEC of the transformed program can be roughly represented as follows:

$$TEC(P') = \sum_{i=1}^{N_{kernel}} (II_i * (TC_i - 1) + Latency_i) \tag{2}$$

where II_i, TC_i, and $Latency_i$ indicate the initiation interval of kernel i, the trip count of kernel i and the latency of kernel i, respectively. As II_i is determined via modulo scheduling involving a time-consuming mapping algorithm [3], we can replace II_i with MII_i in performance profiling. As mentioned in [5], MII_i is determined by the resource-constrained Minimal II (ResMII) and the recurrence-constrained Minimal II (RecMII), i.e., $MII_i = \max(ResMII_i, RecMII_i)$.

B. Builder of 1-D Transformation

As the whole transformation is achieved by performing interleaved statement-wise transformation of splitter and 1-D affine transformation, we first build 1-D affine transformation for a given fusion/fission proposal (i.e., splitters are determined) in this subsection.

1) Validity Constraints: For each dependence e of Write-After-Read, Write-After-Write or RAW [6], given $\vec{i}$ and $\vec{j}$ indicating the iteration vectors of the source statement (S_i) and the target statement (S_j) of e, the 1-D affine transformations are valid only if the following constraint is satisfied:

$$\forall k = 2n+1, n \in [0, m_S) : \phi^k_{S_j}(\vec{j}) \geq \phi^k_{S_i}(\vec{i}) \tag{3}$$

2) Orthogonal Constraints: When the loop is multidimensional, the polyhedral model will find linearly independent affine transformations matching the dimensionality of the statement. We introduce additional constraints to guarantee linear independence from the solutions previously identified. This is accomplished by establishing the orthogonal subspace of the transformation rows obtained (H_S) and ensuring a non-zero component within it for the next solution [5].

$$H^{\perp}_S = I - H^T_S \left(H_S H^T_S\right)^{-1} H_S \tag{4}$$

3) Coefficient Constraints: Since loop flattening is limited to a rectangular iteration domain [4], we need to ensure that the iteration domain maintains a rectangular shape after transformation. Consequently, the coefficient c^k_i of affine transformations in Eqn. 1 should satisfy the constraints as follows:

$$\forall k = 2n+1, n \in [0, m_S) : \sum_{i=1}^{m_S} \left|c^k_i\right| = 1; \forall i, c^k_i \in \{-1, 0, 1\} \tag{5}$$

With this constraint, the affine transformation is simplified to the combination of loop interchange and loop reversal [6], which would greatly reduce the search space.

4) Interchange Priority Constraints: As shown in the motivating example, loop interchange would change the direction of RAW or RAR reuse vectors, determine which reuse could be routed via intra-PEA resource, and finally determine how many memory access operations could be eliminated. Thus, to reduce the search space, we can previously evaluate each loop level assuming it is the innermost loop according to the reuse vectors. Given a specified loop level i, assuming that i is interchanged as the innermost one and E_1^i and E_2^i are the set of dependence (RAW) along the innermost loop and the set of reuse (RAR) along the innermost loop, the i^{th} loop is evaluated as follows:

$$dimScore(i) = \sum_{e \in E_1^i} \frac{w_1}{|\overrightarrow{d(e)}|} + \sum_{e \in E_2^i} \frac{w_2}{|\overrightarrow{d(e)}|} \tag{6}$$

where w_1, w_2, and $\overrightarrow{d(e)}$ indicate the weight of RAW reuse vector, the weight of RAR reuse vector, and the distance of e, respectively. Since RAW reuse vector may have a crucial influence [5] on the recMII, w_1 is set less than w_2 (values of 1 and 2 in the experiments). In this equation, we also note that long distances will decrease the score since the corresponding reuse vector would consume more routing resources in PEA.

After evaluation, the level with the highest score (i^*) is first selected as the innermost one. Consequently, the innermost-loop-specific constraint is represented as follows:

$$\forall k = 2n+1, n \in [0, m_S) : c_{i*}^k = \begin{cases} 1, & k = 2m_s - 1 \\ 0, & k \neq 2m_s - 1 \end{cases} \tag{7}$$

5) Sibling-Loop Eliminating Constraint: Since loop flattening doesn't support sibling loops, for any two statements, S_i and S_j, one of the following constraints should be satisfied.

$$\underbrace{\phi_{S_i}^0 \neq \phi_{S_j}^0}_{fission} \vee \underbrace{\forall k \in [0, 2d_s] : \phi_{S_i}^k = \phi_{S_j}^k}_{fusion},\ d_s = \min(m_{S_i}, m_{S_j}) - 1 \tag{8}$$

where m_{S_i} and m_{S_j} indicate the dimensional of statement S_i and S_j. The first constraint above indicates that S_i and S_j would be distributed apart from the outermost loop while the second constraint above indicates that the two statements would be fused till the common innermost loop. Thus, sibling loops could be eliminated with equation (8).

6) Cost Function for 1-D Affine Transformation: As we traverse the search space formed by the above constraints, the objective function for 1-D affine transformation could be in any form. For simplicity of implementation, we use the same cost function as that in [6].

V. Efficient Solution

In this section, we provide a search-based approach including 4 basic steps: 1) fusion/fission proposal, 2) affine transformation, 3) loop post-processing, and 4) flattened-loop performance profiling. By iteratively performing the 4 steps with Dynamical Programming, we can finally achieve efficient solutions.

Algorithm 1: DP-based Search

Input: G, the DDG of a given nested loop; $realCost[i,j]$, the achieved cost from the i^{th} SCC to the j^{th} SCC obtained from backend attempts.
Output: Output transformation result P_n^*;

1 $D \leftarrow createDAG(G)$;
2 $D' \leftarrow topologicSort(D)$;
3 **for** $1 \leq n \leq |D'|$ **do**
4 **if** $n == 1$ **then**
5 $P_n \leftarrow \{n\}$;
6 **if** $realCost[n,n]$ *is ever updated* **then**
7 $minCost[n] \leftarrow realCost[n][n]$;
8 **end**
9 **else**
10 $minCost[n] \leftarrow perfProfiling(n, n, D')$;
11 **end**
12 **end**
13 **else**
14 $minCost[n] \leftarrow +\infty$;
15 **for** $0 \leq m < n$ **do**
16 **if** $realCost[m+1, n]$ *was ever updated* **then**
17 $tempCost \leftarrow minCost[m] + realCost[m+1, n]$;
18 **end**
19 **else**
20 $p \leftarrow fusionProposal(m+1, n, D')$;
21 $pSeq \leftarrow prioritySort(D', p)$;
22 $subG \leftarrow mDAffineTransform(p, G, pSeq)$;
23 **if** *subG is not empty and has no sibling loop* **then**
24 $subG \leftarrow postProcessing(subG)$;
25 $nCost \leftarrow perfProfiling(m+1, n, subG)$;
26 **end**
27 **else**
28 $pSeq \leftarrow adjustSeq(prioritySeq)$;
29 **if** *pSeq is not empty* **then**
30 **goto** line 22;
31 **end**
32 $nCost \leftarrow +\infty$;
33 **end**
34 $tempCost \leftarrow minCost[m] + nCost$;
35 **end**
36 **if** *tempCost* $\leq$ *minCost[n]* **then**
37 $minCost[n] \leftarrow tempCost$;
38 $P_n \leftarrow \{P_m, \{\cup_{s=m+1}^{n} s\}\}$;
39 **end**
40 **end**
41 **end**
42 **end**

A. Dynamical Programming (DP) based Search

Algorithm 1 presents the DP-based search algorithm. First, it takes the DDG (G) extracted from a loop nest and the achieved real II ($realCost[i,j]$) from backend attempts as inputs. Then, it creates the DAG (D) from the G by merging the statements in the same SCC into a super node and then removing self-to-self edges. Next, it performs a topological sorting on D and generates a sorted DAG (D'). Then, the algorithm performs dynamic programming to explore fusion/fission, keeping the search complexity at $O(N^2)$ (lines 3-42, where N indicates the number of SCCs in the DAG. In the outer loop, for each SCC n, if n is equal to 1, the first SCC in D' is added to P_1, indicating the fusion result so far; if SCC 1 is ever sent to the backend and gets the real cost ($realCost[n,n]$), the minimal cost ($minCost[n]$), i.e, the minimum latency of the first n SCCs when they are merged and flattened, is set to $realCost[n,n]$; otherwise, $minCost[n]$ is determined by performance profiling. If n is greater than 1, $minCost[n]$ is first initialized with $+\infty$. Then, for each SCC m less than n, if

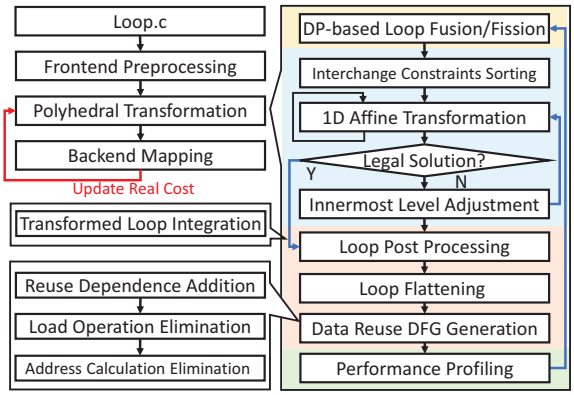

Fig. 3. The overall flow of polyhedral-guided loop flattening.

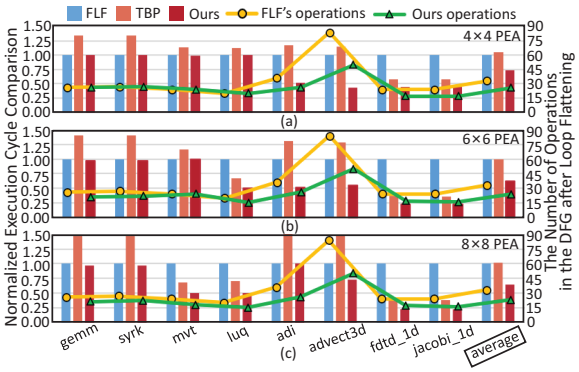

Fig. 4. The normalized (to FLF) execution cycles and the number of operations in the DFG after loop flattening comparison when N=32. (a) Comparison on 4×4 PEA, (b) comparison on 6×6 PEA, and (c) comparison on 8×8 PEA.

$realCost[m+1,n]$ is ever updated by the backend, a temporary cost ($tempCost$) is calculated by adding $minCost[m]$ to $realCost[m+1,n]$. Otherwise, the SCCs from $m+1$ to n are labeled for *loop fusion proposal* and the *multi-dimensional affine transformation* ($mDAfineTransform$) is performed according to the innermost priority ($pSeq$) (lines 20-22). If the converted loop ($subG$) has no sibling loop, *post-processing* is performed to flatten the transformed part into a single-level loop (line 24) and a new cost ($nCost$) is determined by *performance profiling*; otherwise, the innermost layer will be adjusted for the next sub-optimal solution, and the transfer to line 22 will re-execute the polyhedral model transformation until no feasible solution can be found and the innermost layer priority sequence is empty, it is set to $+\infty$; After $nCost$ is updated, the temp cost ($tempCost$) is updated by adding $minCost[m]$ to the new cost. If the temp cost is less than $minCost[n]$, $minCost[n]$ is updated with the temp cost, and the fusion result (P_n) for the previous n SCCs is recorded (line 38).

B. Overall Compilation Flow

Fig. 3 shows the overall flow of our work. First, we convert *C* input into MLIR dialects [11] via front-end preprocessing. Then, we perform polyhedral transformation modified from PluTo [6] to find the flattening-friendly loop structure. The strategy of the DP-based method provides different fusion/fission proposals. For each proposal, we will perform loop transformation by specifying the innermost loop via equations (6) and (7). If no legal solution is found, we will adjust the innermost level and re-perform affine transformation. Otherwise, we will perform loop post-processing to integrate the transformation loops. Next, we flatten the transformed nested loop into one or more single-level loops and generate reused DFG for performance profiling. After generating the final candidate solution, it goes to the backend to perform DFG mapping [3]. If mapping fails, the real cost of the solution will be recorded and sent to polyhedral transformation to update the real cost table and re-generate another transformation for back-end mapping.

VI. Experimental Evaluation

A. Experiment Setup

Benchmark: To verify and evaluate the proposed optimization approach, we extract some performance-critical loops from benchmarks PolyBench 4.2 and Pluto-master [6].

Target architecture: To ensure a comprehensive analysis, we generate three different CGRA configurations: 4×4 PEA, 6×6 PEA, and 8×8 PEA with 4, 6 and 8 memory banks respectively. The PEs in the three configurations are all connected via mesh-like interconnects. For each PE, we use the expanded structure [4] with 4 local registers to support iteration regeneration and data reuse. Then, the performance results are obtained from the established cycle-accurate CGRA simulators.

Compare references: We compare our approach with two references. 1) Fission-based loop flattening (FLF) [4]: this approach explores loop fission and performs loop flattening. If loop fission cannot eliminate sibling loops, it uses partial loop pipelining to get execution cycle. 2) Transformation-based pipeline (TBP) [5]: this approach optimizes partial loop pipelining through extensive loop transformation, where only the innermost loops are pipelined. For a fair comparison, data reuse is also applied to the innermost loop for other references if possible. All mapping approaches run on an Intel 16-core CPU operating at 2.1 GHz with 64 GB of memory.

B. Performance Comparison Over Different CGRA Size

Fig. 4(a), (b) and (c) show the comparison results of the execution cycle and the number of DFG operations after loop flattening with a loop trip count (N) of 32. As the bars shown in Fig. 4(b), our approach can achieve 1.62× and 1.61× speedup on 6×6 PEA as compared to FLF and TBP, respectively. As compared to FLF, it only performs fission exploration to loop structures, leading to few opportunities for data reuse, a big loop body after flattening and poor execution performance. Taking $gemm$, $syrk$ and luq for example, as they include sibling loops, FLF can only distribute these sibling loops apart, losing the chance of RAW data reuse. Meanwhile, our approach can fuse the sibling loop together for RAW data reuse, resulting in fewer operations and smaller IIs. For kernel adi and $advect3d$, as they are both perfectly nested loops, FLF would do nothing on them before loop flattening while our approach can still make loop interchange to reduce the distance of RAR/RAW reuses. Consequently, after data reuse, our approach can achieve smaller DFGs for loop pipelining. As the curves shown in Fig. 4(b), our approach gets 26 and 50 operations in the generated DFGs of kernel adi and $advect3d$ while FLF gets 36 and 84 operations, respectively. For kernel $fdtd\_1d$ and $jacobi\_1d$, as the sibling loops exist in one SCC, FLF fails to distribute them apart and they have to be executed in the partial loop pipelining fashion, leading to significant performance loss.

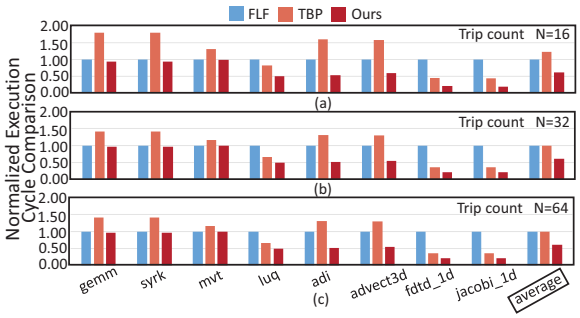

Fig. 5. Execution Cycle comparison on 6×6 CGRA. (a) Execution cycle when N=16, (b) execution cycle when N=32, (c) and execution cycle when N=64.

Meanwhile, our approach eliminates the sibling loop via loop fusion and gets improved performance. As compared to TBP, our improvements mainly come from the full loop pipelining while TBP only performs partial loop pipelining involving significant invocation overheads. Furthermore, without considering reuse optimization in loop transformation, TBP has few opportunities to reduce the DFG size via data reuse.

In Fig. 4(a), our approach shows improved performance as PEA size increases from 4×4 to 6×6. That is because a larger PEA encourages loop fusion without increasing II while FLF doesn't support loop fusion exploration. However, from Fig. 4(c), when the PEA size is increased from 6×6 to 8×8, our approach nearly has no improvement. That is because 8×8 PEA is big enough to hold most DFGs even if the reusable memory accesses are not eliminated.

C. Performance Comparison Over Different Trip-count

Fig. 5(a), (b) and (c) show the execution cycle comparison for the trip count 16, 32 and 64 on 6×6 PEA. Compared to FLF, our method can averagely achieve 1.63×, 1.62×, and 1.61× speedup for the trip count of 16, 32 and 64, respectively. As to TBP, our method can averagely achieve 2.00×, 1.61×, and 1.41× speedup for the trip count of 16, 32 and 64, respectively. As the trip count increases, the speedup over both FLF and TBP slightly decreases. The main reason about FLF is the influence of the kernels including statements of different loop levels, such as kernel $gemm$ and $syrk$. With the growth of trip count, statements with deeper loop levels increasingly dominate the overall execution cycle, limiting the benefits of loop fusion in our approach. The reason for TBP is that the overhead of the prologue and epilogue of innermost-loop pipelining in TBP greatly decreased, resulting in less invocation overhead.

D. Compilation Time

Fig. 6 shows the compilation time on 4×4 PEA, encompassing both loop transformation and mapping. As FLF only supports loop fission, which takes extremely little time on loop transformation, it is not illustrated. As compared to FLF, our method consumes 28.9% total compilation time on a geometric average. For the former 6 kernels, both approaches get the same mapping time since they get the same loop structure for loop flattening. With increased loop transformation time, our approach gets more total compilation time. For other kernels, their DFG sizes are relatively bigger and the mapping time dominates the compilation time. As compared to TBP, our method consumes 33.1% loop transformation time and 2.9% total compilation time on geometric average. The main reason for the reduction in transformation time is that the constraints for sibling loop elimination and innermost loop specification can greatly reduce the search space. Moreover, our approach can effectively reduce the DFG size with data reuse, it also gets less compilation time.

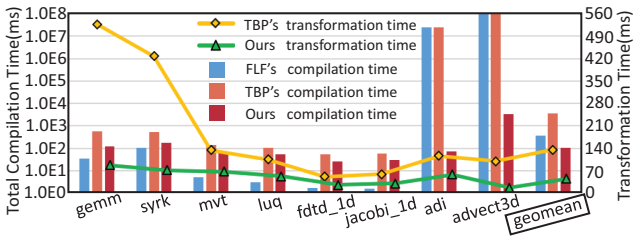

Fig. 6. Total Compilation time and transformation time on 4×4 PEA

VII. Conclusion

Loop flattening is a promising technique to reduce the invocation overhead of nested loops on CGRAs. However, it cannot be directly applied to nested loops containing sibling loops and is prone to form a bigger loop body after flattening. To address this problem, this paper provides a flattening optimization approach using the polyhedral model such that a flattening-friendly loop structure can be found for loop flattening. Experimental results show our approach is effective in improving the performance of nested loops on CGRA.

References

[1] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, "4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[2] R. Zhu, B. Wang, and D. Liu, "RF-CGRA: a routing-friendly CGRA with hierarchical register chains," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2022, pp. 262–267.

[3] S. Dave, M. Balasubramanian, and A. Shrivastava, "RAMP: Resource-aware mapping for CGRAs," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[4] J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for CGRAs," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, 2014, pp. 1–10.

[5] D. Liu, T. Liu, X. Mo, J. Shang, and S. Yin, "Polyhedral-based pipelining of imperfectly-nested loop for CGRAs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[6] U. K. Bondhugula, "Effective automatic parallelization and locality optimization using the polyhedral model," Ph.D. dissertation, The Ohio State University, 2008.

[7] J. Cong, P. Zhang, and Y. Zou, "Combined loop transformation and hierarchy allocation for data reuse optimization," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 185–192.

[8] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li *et al.*, "AKG: automatic kernel generation for neural processing units using polyhedral transformations," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1233–1248.

[9] C. Bastoul, Z. Zhang, H. Razanajato, N. Lossing, A. Susungi, J. de Juan, E. Filhol, B. Jarry, G. Consolaro, and R. Zhang, "Optimizing GPU deep learning operators with polyhedral scheduling constraint injection," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 313–324.

[10] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[11] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral MLIR," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 45–59.