Reducing Memory Access Conflicts with Loop Transformation and Data Reuse on Coarse-grained Reconfigurable Architecture

Yuge Chen*, Zhongyuan Zhao*†, Jianfei Jiang*, Guanghui He*, Zhigang Mao*, Weiguang Sheng*
*Department of Micro/NaNo Electronics, Shanghai Jiao Tong University, Shanghai, China
†School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
Email: †zyzhao.sjtu@gmail.com, *wgshenghit@sjtu.edu.cn

***Abstract*—Coarse-Grained Reconfigurable Arrays (CGRAs) are promising to have low power consumption and high energy-efficiency characteristics as accelerators. Recent years, many research works focus on improving the programmability of the CGRAs by enabling the fast reconfiguration during execution. The performance of these CGRAs critically hinges upon the scheduling power of the compiler. One of the critical challenges is to reduce memory access conflicts using static compilation techniques. Memory accessing conflict brings the synchronization overhead which causes the pipelining stall and reduces CGRA performance. Existing compilers usually tackle this challenge by orchestrating the data placement of the on-chip global memory (OGM) in CGRA to let the parallel memory accesses avoid the bank conflict. However, we find bank conflict is not the only reason that causes the memory access conflicts. In some CGRAs, the bandwidth of the data network between OGM and processing element array (PEA) is also limited due to the low power design principle. The unbalanced network bandwidth loads is another reason that causes memory access conflicts. Furthermore, the redundant data access across iterations is one of the primary causes of memory access conflicts. Based on these observations, we provide a comprehensive and generalized compilation flow to reduce the memory conflicts. Firstly, we develop a loop transformation model to maximize the inter-iteration data reuse of the loops to reduce the memory accessing operations under the software pipelining scheme. Secondly, we enhance the bandwidth utilization of the network between OGM and PEA and avoid the bank conflict by providing a conflict-aware spatial mapping algorithm which can be easily integrated into existing CGRA modulo scheduling compilation flow. Experimental results show our method is capable of improving performance by an average of 44% comparing with state-of-the-art CGRA compiling flow.**

***Index Terms*—CGRA, multi-bank memory, data reuse, spatial mapping**

I. Introduction

CGRAs are computing fabrics which are widely used in computation-intensive applications like video processing [1], digital signal processing [2] and machine learning [3].

There are many representative CGRAs have been proposed, e.g. ADRES [4] and CGRA-ME [5], which provide a typical CGRA structure template. A typical CGRA platform consists of host processor and computing accelerator and the computing accelerator provides high computational efficiency. How CGRA compiler maps computation-intensive kernel (e.g. loop) of the application program onto processing element array (PEA) is still a challenging research topic. Software pipelining technology is widely used in CGRA compilation to exploit the loop and instruction level parallelism. With software pipelining technology, the goal of compiler is to find valid mapping strategy to achieve minimized initiation interval (II). Many existing works [6] [7] present excellent techniques of minimizing II efficiently. However, software pipelining technology increases the data access parallelism requirements of on-chip global memory (OGM), thus the pipelining stall caused by memory access conflicts becomes the bottleneck of CGRA efficiency.

Multi-bank memory technique provides a hardware foundation for solving conflicts problem. The multi-bank on-chip global memory satisfies the demand of multiple memory access by PEA in parallel. However, memory access conflicts problem still remains because of several reasons including accessing bank concurrently, unbalanced bandwidth loads and redundant data access. Most of the existing conflicts reduction techniques [8] [9] tackle this problem by avoiding the bank conflict, but ignore pipelining stall caused by other reasons.

Based on previous analysis, we solve memory access conflicts problem from two aspects: 1) We can consider the bank resource occupation and the bandwidth utilization of the network between OGM and PEA simultaneously to avoid conflicts. 2) We can take advantage of the inter-iteration data reuse of the loops to remove unnecessary memory accessing operations in kernel pipelining. Based on these motivations, the contributions of this paper are summarized as follows:

- We propose a loop transformation model to maximize valid inter-iteration data reuse of the loop. We systematically define data reuse of a perfectly nested loop, and propose a formulated optimization problem of finding feasible loop transformation under dependence constraints to maximize valid inter-iteration data reuse of the loop.
- We propose a heuristic spatial mapping approach called conflict-aware mapping, which considers both the network bandwidth utilization and possible bank conflicts and reduces possible conflicts comprehensively.
- A context modification approach is developed for reducing redundant memory accessing operations. This approach takes valid inter-iteration data reuse into account and chooses modification strategy according to temporal and spatial constraints to eliminate unnecessary global memory

* Weiguang Sheng is corresponding author.

978-3-9819263-5-4/DATE21/©2021 EDAA

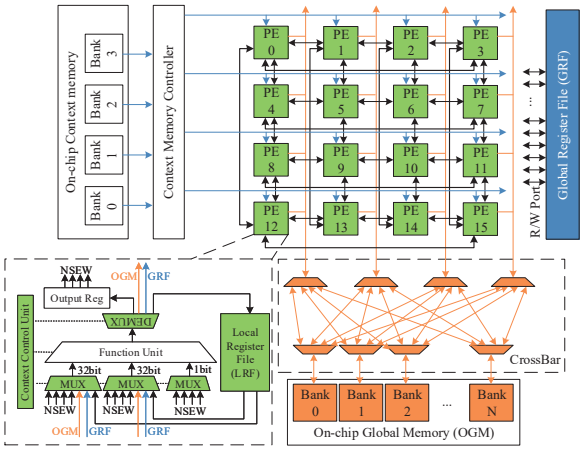

Fig. 1. The system level architecture of the CGRA computing accelerator, including 4×4 PEA and three types memory buffer. PEs access any bank of OGM through the column bus connected.

access operations.

Our method is applied to mapping and context generation process in compiling flow, thus it can be orthogonally combined with existing data placement and modulo scheduling approaches to achieve better performance.

To prove the feasibility of the proposed model, We combine our method with the state-of-the-art scheduling approach in [7] and data partitioning algorithm in [10] to form a complete compiling flow. Experimental results show our method improves performance by an average of 44% comparing with the compiling flow without our method.

II. Background and Related Work

A. The CGRA structure

The system level typical CGRA structure defined by ADRES [4] is shown in Fig. 1, including on-chip global memory (OGM), global register file (GRF), context memory and $N \times N$ PEA. The architecture of PE consists of function unit and register resources. PEs are connected through a torus topology network, and each PE is capable of receiving data from the output register of connected PE, local register file (LRF), GRF and OGM. GRF can be accessed by any PEs through full crossbar fabric. All PEs in the same column are connected to one column bus, which can access any banks of OGM through another full crossbar fabric [9].

During program execution time, the host processor firstly transfers context files generated by CGRA compiler to the context memory, and data for loop program should be placed in multi-bank on-chip global memory. All PEs execute instructions stored in context memory in a synchronous manner. The synchronizer sends messages to PEA to start executing operations of next round only when all working PEs finish operations in this round. A round between two synchronizations is called a control step. However, the actual execution time of a control step increases when memory access conflicts happen, which leads to higher actual II. Most of the existing optimization works based on multi-bank focus on reducing II by solving memory access conflicts.

B. Memory access conflicts

The reasons that cause memory access conflicts come from accessing bank concurrently and unbalanced network bandwidth loads. There will be bank conflict when different PEs access the same bank simultaneously. In low power CGRA, the bandwidth of network between PEA and OGM is limited and unbalanced network bandwidth loads conducts conflicts. We use the network conjunction in Fig. 1 as example, conflicts occur when PEs connected to one column bus access OGM concurrently such as $PE1$ and $PE5$. This kind of conflicts is called column bus conflict. Specifically, each element of GRF is fully connected to PE, thus GRF accesses do not cause conflict.

The unnecessary memory accesses is one of basic reasons that lead to conflicts. Reducing redundant operations during execution may reduce memory accessing conflicts ultimately.

C. Existing conflicts reduction techniques

In order to reducing memory access conflicts during execution, many optimization techniques have been proposed. The array clustering algorithm in [11] is proposed to balance bank loads. But their method only achieves limited performance for the coarse granularity of their array-level data partitioning algorithm. The approach in [12] considers data reuse in loop program. But their method of solving dependence in recurrent loops relies on double-buffering architecture, and the performance of their partitioning algorithm is limited. Data partitioning approaches based on linear transformation in [8] and [13] are proposed to avoid bank conflict during execution. The conflict-free loop mapping strategy [9] proposes a dual-force directed scheduling algorithm to for eliminating access conflicts and minimizing II. However, though existing techniques provide promising methods of reducing bank conflict, the network bandwidth conflict still hasn't been solved. Furthermore, the method of reducing conflicts by eliminating redundant operations is also not considered.

III. Compilation Flow

Fig. 2 shows the overview of the compilation flow. Methods proposed for reducing redundant memory accessing operations are highlighted as blue, and methods proposed for improving network bandwidth utilization and reducing bank conflict are highlighted as orange. The perfectly nested loop program goes through CGRA front-end and generates LLVM IR. Generated IR is fed into loop transformation model for maximum data reuse. The modified loop program goes through data flow graph (DFG) generation, scheduling and data placement processes in turn and generates scheduled DFG ready for mapping. Mapping result is converted to configuration contexts in context generation process. We emphasis on introducing novel methods proposed in this paper.

A. Loop transformation model

Inter-iteration data reuse exists between any two operations in different iterations that access the same data on OGM. According to the order of accessing the data, these two operations form a producer/consumer pair, which is also called reuse pair. Based on the types of producer and consumer, reuse pairs

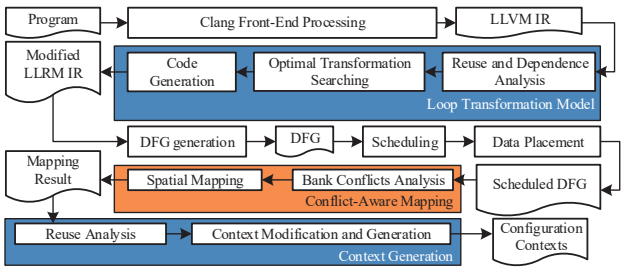

Fig. 2. The overview of the compiling flow, where highlighted processes are methods proposed in this paper

can be divided into four categories, *Load-Load*, *Load-Store*, *Store-Load* and *Store-Store* reuse. For a *Load-Load* or *Store-Load* reuse pair, we use LRF or GRF to buffer the output of producer so that consumer just need to load data from LRF or GRF instead of OGM, thus reducing memory accessing operations. However, as the sizes of GRF and LRF provided by hardware is limited, our compiler should guarantee the reused data can be successfully kept in LRF or GRF during its life time. Therefore, the goal of our loop transformation model is to orchestrate the execution order of the loop iterations using affine transformation method to minimize the life time of the inter-iteration reused data.

We decompose the problem of finding the optimal loop transformation into three sub-problem: Reuse and dependence analysis, optimal transformation searching and code generation. In many cases, there are data reuse opportunity between two operations if they have data dependency. Many existing approaches of dependence analysis like [14] are proposed. Therefore, data reuse can be exploited in a similar way to dependence analysis. Firstly, we describe representation for loop program, data dependence and reuse.

Definition 1. (**Perfectly nested loop**) A perfectly nested loop program is represented as $L = \langle S, \delta, \mathcal{B}, \mathcal{A}, W \rangle$ where

- S is the set of instructions. $S = s_1, s_2, \ldots, s_k$ means there are k instructions in inner loop.
- δ: is the depth of perfectly nested loop.
- $\mathcal{B}(\vec{i}) = B\vec{i} + b$ is an affine expression represent the loop bounds, $\vec{i}$ is a valid loop index if $\mathcal{B}(\vec{i}) \geq 0$.
- $\mathcal{A}_{zr}$ is a memory accessing operation, and $\mathcal{A}_{zr}(\vec{i}) = A_{zr}\vec{i} + a_{zr}$ is an affine expression of this operation, where A_{zr} is δ-dimension vector and a_{zr} is a constant. $\mathcal{A}_{zr}(\vec{i})$ means the address slot in iteration $\vec{i}$ of rth memory accessing operation to array z.
- W_{zr} is true when operation $\mathcal{A}_{zr}$ is a write operation.

Definition 2. (**Data reuse set**) The data reuse set U for loop program L is defined according to (1), only including reuse pairs that no other operation between producer and consumer accesses the same memory slot.

$$\begin{aligned} U = \{ \langle \mathcal{A}_{zr}, \mathcal{A}_{zr'} \rangle \big| (\exists \Delta\vec{i} \in \mathbb{N}^{\delta}, \Delta\vec{i} \neq \vec{0}, \\ \forall \vec{i} \in \mathbb{N}^{\delta} \big| (\mathcal{B}(\vec{i}) \geq 0 \wedge \mathcal{B}(\vec{i} + \Delta\vec{i}) \geq 0) \wedge \\ (\mathcal{A}_{zr}(\vec{i}) - \mathcal{A}_{zr'}(\vec{i} + \Delta\vec{i}) = 0)) \} \end{aligned} \quad (1)$$

Specifically, we use loop program shown in Fig. 3(a) as an example to illustrate these definitions. The dependence set and reuse set of loop are shown in Fig. 3(b). In Fig. 3(c) and (d),

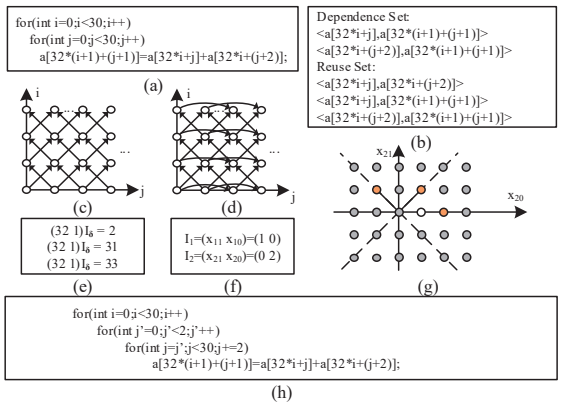

Fig. 3. A example of loop transformation: (a) Original program. (b) Data dependence and reuse set. (c) Program dependence graph. (d) Program reuse graph. (e) Corresponding equations of reuse. (f) Choosen time vectors. (g) Finding the optimal innermost time vector. (h) Transformed program.

each iteration of the loop is represented by nodes, and arrows in Fig. 3(c) represent dependence between iterations and in Fig. 3(d) represent reuse.

Based on data dependencies, legal transformation should be calculated. Different from existing schedule selecting algorithms like Feautrier [15] and PLuTo [16] which focus on maximizing parallelism and minimizing data transformation, this paper proposes a affine transformation selecting algorithm for minimizing iteration interval between each data reuse pair.

Let $\vec{I_m} = \begin{pmatrix} x_{m0} & x_{m1} & \ldots & x_{m\delta} \end{pmatrix}$ be the time vector of loop level m, which represents the index difference between two consecutive iterations in level m. In Fig. 3(a), time vectors of two loop layers $\vec{I_1} = \begin{pmatrix} 1 & 0 \end{pmatrix}$ and $\vec{I_2} = \begin{pmatrix} 0 & 1 \end{pmatrix}$ are shown in Fig. 3(f). A method of selecting valid time vectors recursively satisfies data dependencies is proposed.

Theorem 1: A set M_R contains all dependence vectors in loop. Let $cone(M_R) = \sum_r a_r\vec{i_r} | a_r \geq 0, \vec{i_r} \in M_R$ be the convex cone spanned by the elements of M_R, $cone(M_R)_I$ be the projection of $cone(M_R)$ onto a $\delta - 1$ dimensional hyperplane whose normal vector is $\vec{I}$, and $cone(M_R/\vec{i_r})$ be the convex cone spanned by the elements of M_R except $\vec{i_r}$. Given a k-dimensional data dependence vectors set M_R, $\vec{I} \in \mathbb{N}^k, \vec{I} \neq \vec{0}$ as innermost time vector is valid if (2) or (3) is true:

$$\pm\vec{I} \notin cone(M_R) \quad (2)$$

$$\vec{I} = c\vec{i_r}, c \in \mathbb{R}, 0 < c \leq 1, i_r \in M_R \wedge \vec{i_r} \notin cone(M_R/\vec{i_r}) \quad (3)$$

Proof 1: When (2) is true, there must exist a hyperplane passing through the origin and $cone(M_R)_I$ is on one side of this hyperplane. Thus a $\delta - 1$ dimensional time vector on this side of the hyperplane which satisfies (2) or (3) can be found. Furthermore, when (3) is true, the projection of data dependence $\vec{i_r}$ onto time axis must be positive, and $\pm\vec{I}$ doesn't belong to $cone(M_R/\vec{i_r})$ so that (2) is established.

When search for the optimal transformation, whether a *Load-Load* or *Store-Load* reuse pair is valid depends on the innermost time vector. A reuse pair $\langle \mathcal{A}_{zr}, \mathcal{A}_{zr'} \rangle$ is valid when its corresponding (4) holds. C is a constant threshold value that is used to help selecting the valid reuse pair.

$$A_{zr}\vec{I_\delta} = C\Delta a \quad (4)$$

The optimal transformation searching algorithm will first search the optimal time vector of the innermost loop level and recursively calculates the time vector of the outer level according to the time vector of the inner level. The principle of selecting the optimal time vector is to maximize the number of valid data reuse pairs according to (4) under the data dependence constraints (2) and (3) in theorem 1. The innermost time vector and the reuse vector are collinear is a neccessary condition, thus only vectors which satisfies (5) are in set $Candidate$ to get the optimal solution with less time complexity.

$$Candidate = \{\vec{I} \Big| \exists \vec{u} \in U, \vec{I} = c\vec{u}, c \in \mathbb{R}, 0 < c \leq 1\} \quad (5)$$

The algorithm checks candidate vectors in $Candidate$ to find the optimal innermost time vector, and recursively generates the outer time vectors according to the time vector of the inner level. In code generation process, The loop increment of level m should be set to time vector $\vec{I_m}$.

To illustrate our method, we still take the program shown in Fig. 3(a) as an example. The goal of the optimization problem is to maximize the number of established equations in Fig. 3(e). Nodes in Fig. 3(g) represent candidate innermost time vectors, where feasible solution of optimization problem are nodes highlighted as orange. Fig. 3(f) shows the time vector of each loop layer. Fig. 3(h) shows the transformed program while the innermost loop is split into two layers in order to traverse all iterations on the direction of time vector.

B. Conflict-aware mapping

The overview of conflict-aware mapping processing flow is shown in Fig. 2 where highlighted as orange. Firstly, our method finds out potential bank conflict between memory accessing operations issued in parallel. Linear transformation method is widely used in data placement algorithms such as [9] and [10]. For a memory accessing operation s, the bank index Bk of a memory accessing operation $\mathcal{A}_{zr}$ can be calculated by $Bk_s(\vec{i}) = \mathcal{A}_{zr}(\vec{i}) \ mod \ N_{Bank}$, where N_{Bank} is the number of banks. There will be bank conflict between $\mathcal{A}_{zr}$ and $\mathcal{A}_{z'r'}$ when (6) holds.

$$(\mathcal{A}_{zr}(\vec{i}) - \mathcal{A}_{z'r'}(\vec{i})) \ mod \ N_{Bank} = 0 \quad (6)$$

Equation (6) has the periodicity of N_{Bank}, thus the frequency of possible bank conflict between two operations can be represented by the number of times when this equation is true per N_{Bank} consecutive iterations. Let F_q be the greatest common divisor between each nonzero element in $\Delta A = A_{zr} - A_{z'r'}$ and N_{Bank}. If $(a_{zr} - a_{z'r'}) \ mod \ F_q = 0$ holds, there will be bank conflict between these two operations and F_q represents the conflicts frequency. This value is stored into an adjacent matrix M_B.

If the current initiation interval is II, the process of mapping scheduled DFG $R(V, E)$ to PEA can be formulated as a problem of finding the subgraph of II time-extended CGRA graph (TEC), $R_{II}(V_R, E_R)$. The compiler starts with finding all candidate PE slots for each operation. A candidate PE slots must have enough interconnection resource to place successors and predecessors of node to map. Let v be the node to be mapped, and V_{mapped} be the operations set that have been mapped. $PE(v)$ is a candidate PE slot when $\forall u \in V_{mapped}, (u, v) \in E \ s.t. \ (PE(u), PE(v)) \in E_R \cup (PE(v), PE(u)) \in E_R$.

Algorithm 1 Calculate Conflict Cost

Input: v, p, M_B, $R_{II}(V_R, E_R)$, $ColBW$
Output: Memory access conflicts cost $cost_{conf}(v, p)$

```
1:  ColCst[1...CGRA_X] ← {0,...,0};
2:  MappedOps ← ∅;
3:  V_R[t(v) mod II][p] ← v;
4:  for i = 1 to CGRA_X × CGRA_Y do
5:      BkCst ← 0;
6:      if V_R[t(v) mod II][i] is not free then
7:          u ← the operation on V_R[i];
8:          for each node w in MappedOps do
9:              BkCst ← BkCst + M_B[u][w];
10:         end for
11:         vCol ← i%CGRA_X;
12:         ColCst[vCol] ← Max(ColCst[vCol], BkCst);
13:         if ColCst[vCol] − 1%ColBW = 0 then
14:             ColCst[vCol] ← ColCst[vCol] + 1;
15:         end if
16:         add node u to MappedOps;
17:     end if
18: end for
19: Cost_conflict = Max(ColCst);
```

Each candidate PE slot has cost according to hardware resource utilization and potential bank conflicts. The cost function of mapping operation v to PE p is based on (7), including the hardware resource cost and the conflicts cost, where α is the weight determined empirically. Let N_v be the number of node v's unmapped predecessors and successors and N_p be the the number of free predecessors and successors PEs of p. If $N_v > N_p$, the $cost_{hw}(v, p)$ will be infinity for lacking of enough hardware resource to map remaining operations. If $N_v \leq N_p$, the value of $cost_{hw}(v, p)$ is calculated by $\frac{N_v}{N_p}$.

$$cost(v, p) = cost_{hw}(v, p) + \alpha \times cost_{conf}(v, p) \quad (7)$$

Algorithm 1 shows the process of calculating $cost_{conf}(v, p)$ for $CGRA_X \times CGRA_Y$ PEA, and $ColBW$ is the column bus bandwidth. This algorithm calculates the issue time of each operation with considering bank conflicts and column bus conflict simultaneously (line 9-16), and selects the maximal issue time of each column as the conflict cost of this mapping strategy (line 20). This algorithm comprehensively considers the bank conflict and the conflict caused by unbalanced network bandwidth loads, which achieve better performance than considering partial conflicts discretely.

The compiler chooses candidate PEs with the lowest cost until all operations in $D(V, E)$ are mapped onto TEC $R_{II}(V_R, E_R)$. If there is no candidate PEs of a certain node v, our mapping algorithm tries to backtrack to the previous node u that may lead to an error. Once there is no node to backtrack, the mapping process under current II fails, and the compiler tries to increase II and restart the scheduling and mapping. This method is similar to previous mapping methods proposed in [17] and [7], which have been proven to be low complexity.

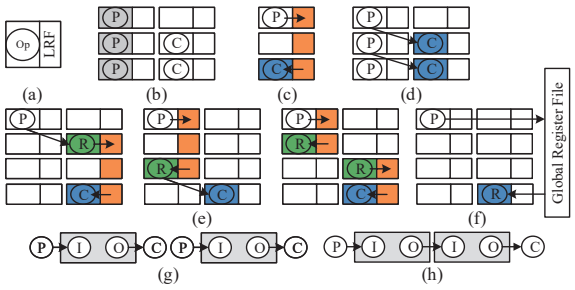

Fig. 4. Different modification strategies for data reuse. (a-f) *Op* represents different operations. Highlighted with blue means converted to routing node, green means adding new routing node, orange means data stored in LRF. (g-h) Modification strategy for accumulation reuse.

C. Context modification algorithm

In context generation, we develop a context modification algorithm to eliminate unnecessary memory accessing operations by exploiting inter-iteration data reuse. The algorithm firstly analyzes valid reuse pairs which satisfy the temporal and spatial constraints, and then different modification strategies are adopted for reducing memory accessing operations. Modification strategies are shown in Fig. 4.

1) Store-Store Reuse: Store-Store reuse is not limited by interval iterations between producer and consumer. Producer of a reuse pair should be eliminated. Fig. 4(b) shows the modification strategy for Store-Store reuse.

2) Load-Load and Store-Load Reuse: For Load-Load and Store-Load reuse, interconnection resources or register resources should be used to transfer data from producer to consumer. Given a data reuse pair $r = \langle p, c, d \rangle$, where p is producer, c is consumer and d is the interval iterations between p and c. Suppose the operation p is mapped on PE $p(p)$ at $t(p)$ time slot in software pipelining, then the interval control steps $I(r)$ between p and c can be calculated by $I(r) = d \times II + (t(c) - t(p))$.

If $I(r) \leq 0$, the reuse pair is invalid. When $I(r) > 0$, spatial constraints of this reuse pair should be checked. Let $Con(r)$ be true when $p(p)$ and $p(c)$ are interconnected. Fig. 4(d) shows the chosen strategy when $I(r) \leq 1$ and $Con(r) = true$, where consumer of reuse is converted to routing operation accessing data from output register of producer.

When $I(r) > 1$, register resources must be used. Fig. 4(c) shows the condition when $p(p) = p(c)$. If remaining space of LRF $N_L^{p(p)}$ on PE $p(p)$ satisfies $N_L^{p(p)} \geq (I(r) + II - 1)/II$, the reused data should be stored into LRF and consumer is converted to routing operation fetching data from LRF. However, when $I(r) > 1, Con(r) = true, p(p) \neq p(c)$, extra routing nodes need to be added for data movement. Suppose there are two null operations np and nc where $p(np) = p(p)$, $p(nc) = p(c)$ and $(t(nc) - t(np))\ mod\ II = 1$, reused data will be moved by these two routing nodes through LRF. It is worthy to note that producer and consumer can also be chosen as np and nc. Fig. 4(e) shows these situations, where $N_L^{p(p)} \geq (I(< p, op, d >) + II - 1)/II$ and $N_L^{p(c)} \geq (I(< oc, c, d >) + II - 1)/II$ should be satisfied.

If none of the above strategies are chosen and the remaining GRF space N_G should satisfies $N_G \geq (I(r) + II - 1)/II$, GRF will be used for storing data temporarily. The modification strategy of this situation is shown in Fig. 4(f).

3) Accumulation Reuse: Given a Store-Load reuse pair $r = \langle p, c, d \rangle$, if $r' = \langle c, p, d \rangle$ is a Load-Store reuse pair, we call r an accumulation reuse. The modification strategy for accumulation reuse is shown in Fig. 4(g) and Fig. 4(h) where the gray blocks represent operations set which contains all operations except memory accessing operations. Nodes labeled I and O are operations that are directly connected to load and store operations, called input operation and output operation. The output operation sends data to input operation based on Store-Load reuse modification strategies and the memory accessing operations during the accumulation should be converted to null operations. This strategy can also reduce the lowerbound of II limited by data dependencies.

IV. Experimental results

A. Methodology

To show the performance of our method, we implement a complete compiling flow based on LLVM 8.0.0 platform, and a CGRA simulator is developed to evaluate the efficiency. We compare the performance with the compiling flow without our methods (called THP+DP) to illustrate the performance improvement when orthogonally combining our work with existing works focus on data partitioning and scheduling. We choose the scheduling approach in [7] which is proven to be more efficient than REGIMap [6] and RAMP [17]. And we choose data partitioning algorithm in [10]. The data placement and scheduling algorithms in [9] are also excellent methods. But their dual-forced scheduling takes implicit routing nodes to transfer data, which is different from our architecture.

We select 22 computation-intensive kernels from existing benchmarks including EEMBC, Polybench [18] and Machsuite [19], and others are derived from digital signal processing, computer vision and dynamic programming.

B. Performance

1) Operation reduction analysis: We make a comparison on the number of times PEA accesses memory in loop pipelining between THP+DP and compiling flow with our work. Table I shows the amount of memory accessing operations comparison. Compared to the compilation flow without our operations reduction algorithm, our work is capable of achieving 55.4% operations reduction in average.

There are 3 kernels *aes3*, *bezier1*, *strassen1* should be noticed. Potential data reuse is explored by loop transformation model and the number of data reuse pairs between two consecutive iterations in these three kernels increase from $0, 1, 14$ to $4, 9, 23$. In addition, other kernels achieve the maximum data reuse in their default states thus they are not changed by loop transformation model.

2) Memory access conflicts comparison: To illustrate the ability of our method to reduce memory conflicts from different sources, Normalized memory access conflicts from network bandwidth limitation is shown in Fig. 5. Theoretical minimum

TABLE I
THE MEMORY ACCESSING TIMES COMPARISON

Kernel	Origin	Our	Kernel	Origin	Our
aes3	1680	1400	harris	3584	1664
aes5	2240	2240	laplace	3844	3844
bezier1	4500	1800	MaxSubString1	2560	516
conven1	1024	643	montecarlo	2560	576
correlation	768	288	montecarlo1	896	276
dp4	640	361	mvt	2048	1040
fil2	2048	2048	nbody	3584	1184
fir3	2592	1350	strassen1	7168	2080
gaosi	1200	384	symm	768	768
gemm1	8192	4352	unstructured2	1536	1536
gesummv	1280	560	wavelet	4096	1030

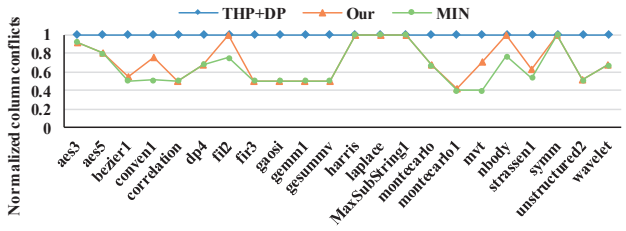

Fig. 5. Relative memory access conflicts comparison

conflicts (called MIN) are calculated by the amount of memory accessing operations per control step divided by the number of column buses. Our method significantly reduces the number of column bus conflict, and conflicts are reduced to theoretical minimum conflicts in most kernels.

For kernels like *conven1*, *fil2*, *mvt* and *nbody*, our method is inefficient. The reasons of minimizing conflicts failure are different among those kernels. For *fil2* and *nbody*, difficulty in finding global optimum solution during mapping process induces unbalanced network bandwidth loads. For *conven1* and *mvt*, though the number of conflicts is reduced through context modification algorithm, operations that produce column bus conflict still remain. Remapping after context modification may cope with this problem but it leads to high complexity.

3) Runtime comparison: Fig. 6 shows the performance comparison on 4×4 CGRA with LRF=2 at each PE and GRF=16. The performance of each kernel is represented by normalized execution times. The *CAMap* means only conflict-aware mapping strategy is adapted and the *CAMap+OpRd* is the compiling flow with our conflict-aware mapping and context modification algorithm. With our method, the compiling flow achieves a 44% improvement in performance on average.

Noticed that *CAMap+OpRd* is inefficient in *fil2* and *symm*. Not containing any redundant memory accessing operations and having achieved balanced network bandwidth loads without our method cause poor performance of these kernels.

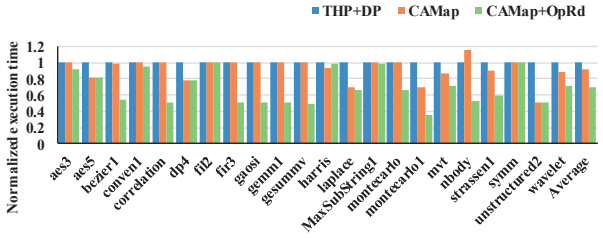

Fig. 6. The execution time comparison using different compiling flow for 4×4 CGRA with LRF=2 at each PE and GRF=16.

V. CONCLUSION

In this paper, we provide a comprehensive and generalized compiling flow to reduce memory access conflicts. We develop a loop transformation model and a context modification algorithm to reduce redundant operations, and a heuristic mapping algorithm is proposed to improve network bandwidth utilization. Our work can be orthogonally combined with existing approaches to achieve better performance. Experimental results illustrate that our method can improve the performance significantly comparing with state-of-the-art CGRA compiling flow.

REFERENCES

[1] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[2] O. Akbari, M. Kamal, A. Afzali-Kusha, M. Pedram, and M. Shafique, "Px-cgra: Polymorphic approximate coarse-grained reconfigurable architecture," in *DATE*, 2018, pp. 413–418.

[3] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *ASPLOS*, 2018, p. 461475.

[4] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the adres coarse-grained reconfigurable array," pp. 1–13, 2007.

[5] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "Cgra-me: A unified framework for cgra modelling and exploration," in *ASAP*, 2017, pp. 184–189.

[6] M. Hamzeh, A. Shrivastava, and S. Vrudhula, "Regimap: Register-aware application mapping on coarse-grained reconfigurable architectures (cgras)," in *DAC*, 2013, pp. 1–10.

[7] Z. Zhao, W. Sheng, Q. Wang, W. Yin, P. Ye, J. Li, and Z. Mao, "Towards higher performance and robust compilation for cgra modulo scheduling," *TPDS*, vol. 31, no. 9, pp. 2201–2219, 2020.

[8] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *FPGA*, 2014, pp. 199–208.

[9] S. Yin, X. Yao, T. Lu, D. Liu, J. Gu, L. Liu, and S. Wei, "Conflict-free loop mapping for coarse-grained reconfigurable architecture with multi-bank memory," *TPDS*, vol. 28, no. 9, pp. 2471–2485, 2017.

[10] Z. Zhao, Y. Liu, W. Sheng, T. Krishna, Q. Wang, and Z. Mao, "Optimizing the data placement and transformation for multi-bank cgra computing system," in *DATE*, 2018, pp. 1087–1092.

[11] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Operation and data mapping for cgras with multi-bank memory," in *Acm Sigplan/sigbed Conference on Languages*, 2010, pp. 17–26.

[12] Y. Kim, J. Lee, A. Shrivastava, J. W. Yoon, D. Cho, and Y. Paek, "High throughput data mapping for coarse-grained reconfigurable architectures," *TCAD*, vol. 30, no. 11, pp. 1599–1609, 2011.

[13] Shouyi Yin, Zhicong Xie, Chenyue Meng, Leibo Liu, and Shaojun Wei, "Multibank memory optimization for parallel data access in multiple data arrays," in *ICCAD*, 2016, pp. 1–8.

[14] W. Pugh and D. Wonnacott, "Eliminating false data dependences using the omega test," *Acm Sigplan Notices*, vol. 27, no. 7, pp. 140–151, 1992.

[15] P. Feautrier, "Some efficient solutions to the affine scheduling problem. part i one-dimensional time," *International Journal of Parallel Programming*, vol. 21, 08 1996.

[16] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," vol. 43, 05 2008.

[17] S. Dave, M. Balasubramanian, and A. Shrivastava, "Ramp: Resource-aware mapping for cgras," in *DAC*, 2018, pp. 1–6.

[18] L.-N. Pouchet *et al.*, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[19] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *International Symposium on Workload Characterization (IISWC)*, 2014, pp. 110–119.