This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

SAT-Based Mapping of Data-Flow Graphs onto Coarse-Grained Reconfigurable Arrays

Yukio Miyasaka[1], Masahiro Fujita[2], Alan Mishchenko[1], and John Wawrzynek[1]

[1] UC Berkeley, CA, USA,
yukio_miyasaka@berkeley.edu
[2] The University of Tokyo, Tokyo, Japan

Abstract. Recently, it has been common to use parallel processing for machine learning. CGRAs are drawing attention in terms of reconfigurability and high performance. We propose a method to map data-flow graphs onto CGRAs by SAT solving. The proposed method can perform the automatic transformation which changes the order of operations in data-flow graphs to obtain more efficient schedules. It also accommodates mapping of multi-node operations like MAC operation. We have solved mapping problems of matrix-vector multiplication. In our experiment, a SAT solver outperformed an ILP solver. Our method successfully processed a data-flow graph of more than a hundred nodes. The automatic transformation under the associative and commutative laws was not as much scalable but successfully reduced the number of cycles, where the XBTree-based method worked faster than the enumeration-based method. As another direction, we tried to optimize a CGRA architecture according to a data-flow graph and were able to reduce its PEs and connections through incremental SAT solving.

Keywords: SAT problem, mapping, data-flow graph, CGRA

1 Introduction

Neural networks are used for machine learning in many fields including image recognition [1]. The calculation of neural network involves numerous MAC operations, and there have been many accelerators developed. For example, TPU [2] has a square mesh of MAC operation units and efficiently performs matrix multiplication in a pipelined manner.

On the other hand, fabricating an ASIC for each application is costly, and reconfigurable devices such as FPGAs attract attention these days. A CGRA (Coarse-Grained Reconfigurable Array) is a reconfigurable device that consists of ALU-like units, whereas an FPGA consists of LUTs. It has been shown that CGRAs achieve higher performance and better energy-efficiency than FPGAs for domain-specific applications [3].

This paper proposes a SAT-based method for mapping data-flow graphs onto CGRAs. The optimality of the obtained schedule in terms of the number of cycles can be proved by checking the mapping problem with one less number of cycles

is unsatisfiable. The proposed method can apply transformations to data-flow graphs during the mapping process by using XBTrees [4] that can implicitly express all possible orders of operations under the associative and commutative laws. Our method also supports mapping of multi-node operations.

In the experiments, we compared the XBTree-based transformation method with the enumeration-based method on mapping problems of matrix-vector multiplication. The XBTree-based method was able to solve the problems that cannot be solved in time by the enumeration-based method. Furthermore, as an additional attempt, we tried to optimize a CGRA architecture according to a data-flow graph. Through iterative synthesis with incremental SAT solving [5], we were able to remove several PEs and connections in a 3×3 square mesh architecture without degrading the performance for an AES data-flow graph.

This paper is organized as follows. Section 2 reviews related work and contrasts our approach. Section 3 explains the basics of SAT problem. Section 4 defines the mapping problem, explains the core of our SAT-based mapping method, and compares a SAT solver and an ILP solver. Section 5 explains the enumeration-based transformation methods proposed in our previous work and shows mapping results for sparse matrix multiplication as an example. Section 6 proposes the XBTree-based transformation method and compares it with the enumeration-based method on mapping of matrix vector multiplication. Section 7 explains a CGRA optimization method and shows a result for a mesh CGRA and an AES data-flow graph. Section 8 concludes the paper.

2 Related work

There have been many studies on mapping onto CGRAs. A study [6] proposed MRRG (Modulo Routing Resource Graph), where the computational resources are duplicated by the number of cycles as the time-frame expansion, and performed simulated annealing under the law of causality. Recent studies replaced simulated annealing by ILP (Integer Linear Programming), which can prove the possibility of mapping in the given number of cycles [7]. We use a SAT solver instead of an ILP solver because the SAT solver worked faster than the ILP solver as shown in our preliminary experiment.

The studies above and others, as far as we know, did not modify the data-flow graphs generated by architecture-agnostic compilers. A study [8] proposed an approach using an encoding like an SMT (Satisfiability Modulo Theories) solver to map linear functions and optimize them during the mapping process, but it was limited to linear functions. Our method can automatically optimize data-flow graphs according to the architectures of CGRAs.

Some studies [9] mapped data-flow graphs at the level of assembly language, but we target high-level data-flow graphs where nodes correspond to arithmetic operations [10] or subroutines. Such a high-level description makes it easy for the mapper to optimize the calculation by changing the order of operations.

Our previous work [11] used a table to enumerate all possible orders of operations for the automatic transformation. That caused a combinational explosion

where the number of intermediate values increases exponentially over the number of contiguous associative (and commutative) operations. To improve the scalability, this work adopts XBTrees and sorters. An XBTree is a binary tree with exchangers and can implicitly express all possible structures of binary trees with a specific number of leaf nodes. Besides, this paper extends the mapping method to optimization of CGRA architectures.

3 SAT problem

A SAT (Satisfiability) problem is a problem to find an assignment to the variables that makes the given logic formula evaluate to true. If there is such an assignment, the logic formula is satisfiable (SAT); otherwise, it is unsatisfiable (UNSAT). The modern SAT solvers take a CNF (Conjunctive Normal Form) formula as an input. A CNF formula is a conjunction of clauses, a clause is a disjunction of literals, and a literal is a boolean variable or its negation.

For example, the CNF formula (1) consists of clauses a, $(\overline{a} \vee \overline{c})$, and $(\overline{a} \vee b \vee c)$, where a, $\overline{a}$, b, c, and $\overline{c}$ are the literals, and a, b, and c are the variables. This CNF formula is satisfiable with the assignment $(a, b, c) = (\text{true}, \text{true}, \text{false})$. If we add a clause $\overline{b}$ to it as shown in the CNF formula (2), it becomes unsatisfiable.

In many applications, we need to impose a constraint that more than K of the specified literals cannot be true at the same time. (3) is a typical representation of the constraint on N literals $\{a_0, a_1, ..., a_{N-1}\}$ with integer addition $(+)$, where it is assumed that true is 1 and false is 0 as an integer. This constraint is called an at most K constraint. In this paper, we used the bimander encoding [12] (with two literals in each group) if $K = 1$, and the sequential unary counter encoding [13] otherwise.

$$a \wedge (\overline{a} \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \tag{1}$$

$$a \wedge (\overline{a} \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge \overline{b} \tag{2}$$

$$a_0 + a_1 + ... + a_{N-1} \leq K \tag{3}$$

4 Mapping problem

4.1 Data-flow graph

A data-flow graph in this paper is a directed acyclic graph that represents calculation as trees of operations. We call the leaf nodes as input-nodes as they correspond to the input variables of the calculation. The internal nodes are called operator-nodes, and each of them represents one operation. The edges pass the values of nodes: the input variables of input-nodes, and the results of the operations of operator-nodes. An operator-node uses the received values as the operands of the operation. The incoming edges to an operator-node are

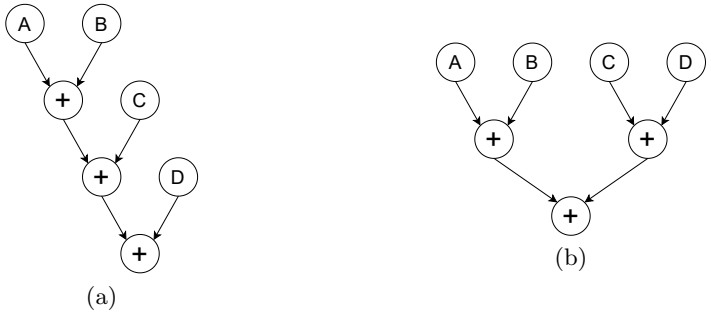

Fig. 1. Data-flow graphs for sum of four variables: A, B, C, and D.

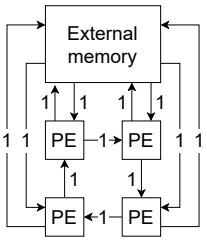

Fig. 2. A CGRA consisting of four PEs connected in a one-way ring.

labeled to specify the order of the operands when the operation is not commutative. The operator-nodes whose results are the outputs of the calculation are also called as result-nodes.

A data-flow graph is not a canonical representation. For example, there are multiple data-flow graphs for sum of four variables as shown in Fig. 1. The order of the operations is fixed in a data-flow graph even if it is arbitrary in nature. In this case, one can be transformed to the other under the associative law.

4.2 CGRA

We represent a CGRA as a directed graph. We call nodes and edges in CGRAs as components and paths respectively to distinguish those from those in data-flow graphs. There are three kinds of components: PE (Processing Element), memory, and external memory. Each PE has a given number of operation-units and registers. A memory holds the values received from other components, and its size is unlimited unless specified. For each memory, a user can specify the input variables that are stored in the memory in advance of the calculation, and even use it as a ROM by removing its incoming edges. The external memory is a memory that supplies the specified input variables and collects the outputs. It can store and pass intermediate values by a user setting. Each path is labeled by the number of values it can pass in one cycle. An example of CGRA is shown at Fig. 2.

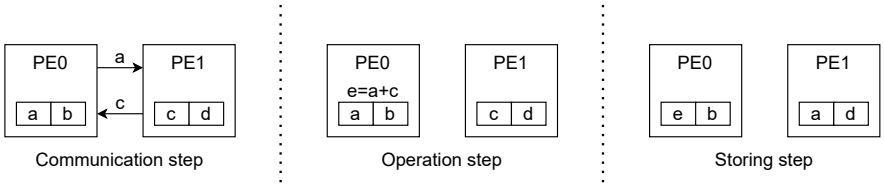

Fig. 3. An example of cycle.

We assume that all components in a CGRA are synchronized. We divide one cycle into three steps: communication step, operation step, and storing step as shown in Fig. 3. First, some of the values in the registers in PEs and the values in memories are passed to other components in the communication step. Next, each operation-unit performs at most one operation in the operation step. An operation-unit can use as the operands the values in the registers in the same PE and the values communicated to that PE in that cycle. Finally, in the storing step, the registers and memories store the values. A register stores one value from the values in the registers in the same PE, the results of the operations in that PE, and the values communicated to that PE in that cycle. A memory, if the number of residing values exceeds a specified limit, selects the values to discard.

4.3 SAT-based mapping

We create a CNF formula using three kinds of boolean variables shown below. Let i, j, k, and h be integers. We number (zero-based) the nodes in a data-flow graph and the components and the paths in a CGRA. Let node i denote the i-th node, component j denote the j-th component, path h denote the h-th path, and cycle k denote the k-th cycle. The range of k is from 0 to $N-1$, where N is the number of cycles. In the following, node i also means the value of the node.

- $X_{i,j,k}$ ... node i exists in component j at the end of cycle k
- $Y_{i,h,k}$ ... node i is communicated in path h at cycle k
- $Z_{i,j,k}$ ... node i is calculated in component j at cycle k

$X_{i,j,k}$ means node i is stored (in the registers) in component j at cycle k.

The CNF is composed of the following clauses. Let H_j denote the set of incoming paths to component j, s_h denote the component at the origin of path h, component e denote the external memory, D_i denote the set of the nodes which are the operands (the nodes at the origins of the incoming edges) of node i, and O denote the set of the result-nodes.

1. $X_{i,j,0}$ for $\forall(i,j)$ where component j is a memory or external memory storing node i, which is an input-node, in advance of the calculation
2. $\neg X_{i,j,0}$ for $\forall(i,j)$ where the condition above is not met
3. $X_{i,e,N-1}$ for $\forall i \in O$

4. $\neg X_{i,j,k} \vee X_{i,j,k-1} \vee \bigvee_{h \in H_j} Y_{i,h,k} \vee Z_{i,j,k}$ for $\forall (i, j, k \neq 0)$
5. $\neg Y_{i,h,k} \vee X_{i,s_h,k-1}$ for $\forall (i, h, k \neq 0)$
6. $\neg Z_{i,j,k} \vee X_{d,j,k-1} \vee \bigvee_{h \in H_j} Y_{d,h,k}$ for $\forall d \in D_i$, for $\forall (i, j)$ where node i is an operator-node and component j is a PE, for $\forall k \neq 0$
7. $\neg Z_{i,j,k}$ for $\forall (i, j)$ where the condition above is not met, for $\forall k \neq 0$
8. At most K constraint on $\{X_{i,j,k}$ for $\forall i\}$ where K is the number of registers in component j, for $\forall j$ where component j is a PE, for $\forall k$
9. At most K constraint on $\{Y_{i,h,k}$ for $\forall i\}$ where K is the label of path h, for $\forall (h, k \neq 0)$
10. At most K constraint on $\{Z_{i,j,k}$ for $\forall i\}$ where K is the number of operation-units in component j, for $\forall j$ where component j is a PE, for $\forall k \neq 0$

The clauses 1 and 2 set the initial condition and the clause 3 sets the condition at the end of the computation. The clause 4 imposes the constraints that the existence of a node in a component infers its existence in the component at the previous cycle, its communication to the component at the same cycle, or its calculation in the component at the same cycle (if the node is an operator-node and the component is a PE). The clause 5 imposes the constraint that the communication of a node in a path infers its existence in the component at the origin of the path. The clauses 6 and 7 impose the constraint that the calculation of a node in a component infers that the node is an operator-node, the component is a PE, and all of its operands are available there. The clause 8 limits the number of nodes in a PE to the number of registers in the PE, the clause 9 limits the number of nodes communicated in a path to the label of the path, and the clause 10 limits the number of the operations performed simultaneously in a PE to the number of operation-units in the PE. Additional clauses are necessary if a user limits the size of memories or forbids the external memory to store and pass the intermediate values.

It is assumed that an operation-unit can process one operator-node in a cycle. On condition that an operation-unit can process two or more operator-nodes at the same time, MAC operation for example, we need to use a different formulation. Fortunately, the formulation for the enumeration-based transformation method also works for that purpose.

The formulation above does not consider pipelining. To be pipelined, a schedule must not use the same resource in multiple cycles that are congruent modulo T, where T is the number of contexts. This constraint can be imposed by modifying the clauses 8, 9, and 10. Let t be an integer. The modification to the clause 8 is shown below, where the changes are written in a bold font. The same modification must be applied to the clauses 9 and 10.

8. At most K constraint on $\{X_{i,j,k}$ for $\forall i$**, for** $\forall \boldsymbol{k}$ **where** $\boldsymbol{k \bmod T = t}\}$ where K is the number of the registers in component j, for $\forall j$ where component j is a PE, for $\forall \boldsymbol{t} \in \boldsymbol{[0, T-1]}$

4.4 Preliminary experiment

As a preliminary experiment, we compared a SAT solver (KISSAT) and an ILP solver (CPLEX v20.1), each running in a single thread. In the ILP problems, the

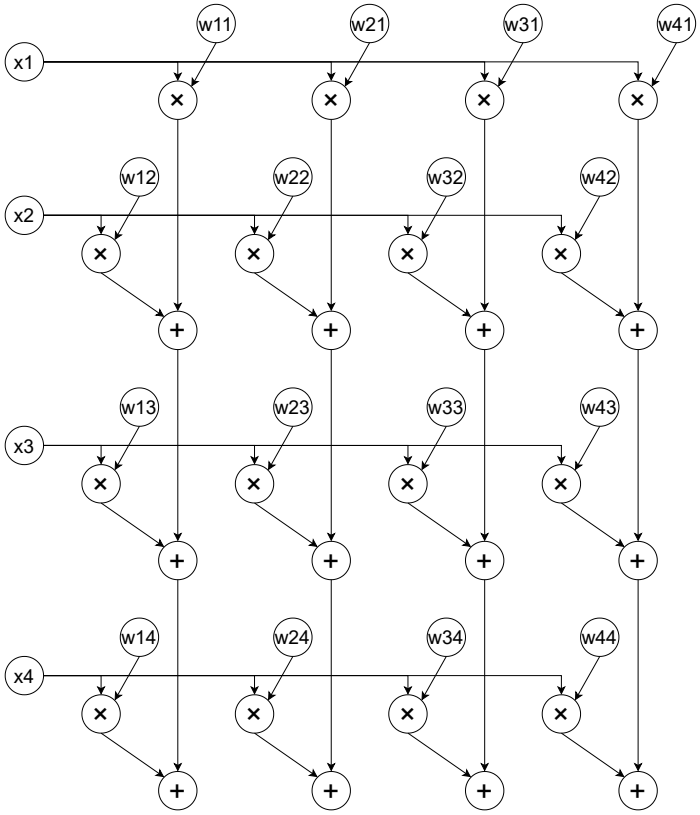

Fig. 4. A data-flow graph of 4×4 matrix-vector multiplication.

at most K constraints were directly expressed without encoding. A parameter "emphasis mip" was set at one for the ILP solver to focus on the satisfiability.

We mapped a data-flow graph of 4×4 matrix-vector multiplication shown at Fig. 4 and a CGRA consisting of four PEs connected in a one-way ring shown at Fig. 2. The number of operation-units and the number of registers in each PE were set at 1 and 2 respectively. No pipelining was done. The external memory was not allowed to store the intermediate values.

The runtime is shown in Table 1. The mapping problem was UNSAT when the number of cycles was 9 and SAT when it was 10. It means that 10 is the minimum possible number of cycles. The SAT solver solved the problems more than a hundred times faster than the ILP solver. The runtime of ILP solver is reasonable because several studies [7, 14, 15] showed that an ILP solver got timeout (one day) just in mapping a data-flow graph consisting of dozens of nodes.

Table 1. The results and runtime in seconds for mapping 4×4 matrix-vector multiplication onto the CGRA of four PEs in a one-way ring.

Cycle	Result	Runtime	
		SAT solver	ILP solver
9	UNSAT	8.9	3567.8
10	SAT	1.2	767.1

5 Enumeration-based transformation

5.1 CNF formulation

The enumeration-based automatic transformation method was proposed in our previous work [11]. It enumerates all possible operations that calculate each node and modifies the CNF formula so that the node can be calculated by any of those operations. For example, the value $A + B + C$ can be calculated as $(A+B)+C$, $(B+C)+A$, or $(C+A)+B$, but one data-flow graph can represent only one of them. The method enumerates all of them from one data-flow graph by traversing it and uses the modified CNF formula where any of them can be used in mapping.

The enumeration of all possible operations under the associative and commutative laws is performed as follows. First, we create cluster-nodes by merging all contiguous operator-nodes of the same associative operator. An example is shown at Fig. 5. Next, we enumerate all candidates for the last operation for each cluster-node, taking the commutativity of the operator into account. For example, for multiplication of four variables, there are seven candidates: four candidates are the multiplication between one variable and the product of the other three variables, and three candidates are the multiplication between the product of two variables and the product of the other two variables. Finally, for each intermediate value we encountered, we create a new node, called an intermediate-node, and enumerate all candidates for its last operation. This final step is recursively done until every intermediate value corresponds to one intermediate-node. Intermediate-nodes are virtual and not located in data-flow graphs. We create a table, as shown in Table 2 for example, to avoid duplication of intermediate-nodes.

The CNF formula is modified to enable each node to be calculated by any of the enumerated candidates. Here, the cluster-nodes and intermediate-nodes are treated as operator-nodes. Let l be an integer, L_i be the number of candidates to calculate node i, and $D_{i,l}$ be the set of the nodes which are the operands in the l-th candidate to calculate node i. We use a new kind of boolean variable, $Z_{i,j,k,l}$, which means all the operands in the l-th candidate to calculate node i are available in component j at cycle k. The clause 6 is replaced by the following clauses:

6.1. $\neg Z_{i,j,k,l} \vee X_{d,j,k-1} \vee \bigvee_{h \in H_j} Y_{d,h,k}$ for $\forall d \in D_{i,l}$, for $\forall l \in [0, L_i - 1]$, for $\forall (i, j)$ where node i is an operator-node and component j is a PE, for $\forall k \neq 0$

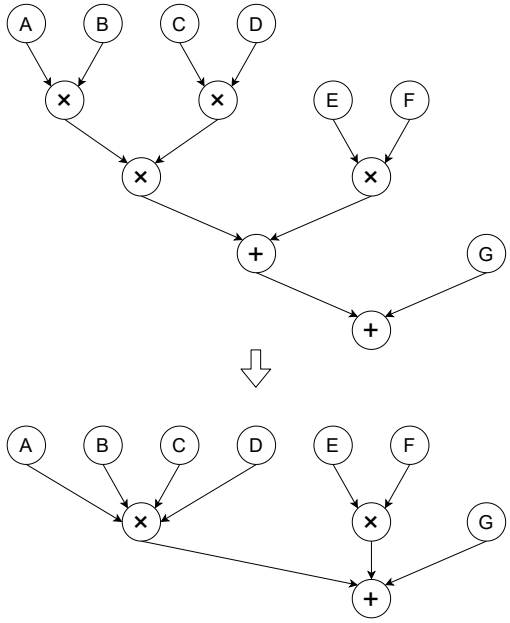

Fig. 5. Creation of cluster-nodes by merging contiguous operator-nodes of the same associative operator.

6.2. $\neg Z_{i,j,k} \vee \bigvee_{l \in [0, L_i - 1]} Z_{i,j,k,l}$ for $\forall (i, j)$ where node i is an operator-node and component j is a PE, for $\forall k \neq 0$

This formulation also makes mapping of multi-node operations possible. A multi-node operation is an operation that simultaneously processes multiple nodes in data-flow graphs. For example, MAC operation, which processes contiguous multiplication and addition, can be mapped as follows. For each node i that is an operator-node of addition, for each candidate to calculate node i, let node a be the first operand and node b be the second operand. If node a is an operator-node of multiplication, for each candidate D (a pair of operands) to calculate node a, we add $D \cup \{b\}$ to the candidates to calculate node i. The same for node b. By doing this, we can map MAC operation using the same CNF formula. Note that the addition of candidates is actually performed after all new candidates are enumerated for all nodes and for all multi-node operations in order to prevent the new candidates from interfering each other,

5.2 Example: sparse matrix multiplication

As an example, we synthesized sparse matrix multiplication algorithms based on TPU [2] using our mapping method. We can obtain the fastest algorithm exploiting the sparsity for each sparse matrix. Fig. 6 shows the original algorithm for $A = W \cdot X$ where W, X, and A are 3×3 matrices. Each element of W is

Table 2. A table from a node to its candidates after processing a cluster-node of multiplication of four variables: A, B, C, and D.

Node	Candidates
$A \times B \times C \times D$	$(A \times B \times C) \times D$, $(A \times B \times D) \times C$, $(A \times C \times D) \times B$, $(B \times C \times D) \times A$, $(A \times B) \times (C \times D)$, $(A \times C) \times (B \times D)$, $(A \times D) \times (B \times C)$
$A \times B \times C$	$(A \times B) \times C$, $(A \times C) \times B$, $(B \times C) \times A$
$A \times B \times D$	$(A \times B) \times D$, $(A \times D) \times B$, $(B \times D) \times A$
$A \times C \times D$	$(A \times C) \times D$, $(A \times D) \times C$, $(C \times D) \times A$
$B \times C \times D$	$(B \times C) \times D$, $(B \times D) \times C$, $(C \times D) \times B$
$A \times B$	$A \times B$
$A \times C$	$A \times C$
$A \times D$	$A \times D$
$B \times C$	$B \times C$
$B \times D$	$B \times D$
$C \times D$	$C \times D$

assigned to one PE. The elements of X are passed to the right, and the partial sums are passed down in the figure. Each PE multiplies a received element of X and the assigned element of W and adds the result and a received partial sum. It takes eight cycles to calculate A, where the number of contexts is 3. The figure also shows the beginning of the next matrix multiplication $B = W \cdot Y$.

We solved the problems mapping sparse matrix multiplication where some elements of W are zero and the corresponding multiplications can be skipped. We used a CGRA of the same topology with each path labeled by one. The number of operation-units and the number of registers in each PE are 1 and 2 respectively, and MAC operation was enabled. The inputs come from the external memory, to which the outputs are returned. We put a ROM for each PE and connected the ROM to the PE. At most one none-zero element of W can be assigned to each ROM. Because of the automatic transformation, permutations of rows and columns of W can be done automatically with swapping output-columns and input-rows, so we only have to consider permutationally inequivalent matrices. There are 36 permutationally inequivalent matrices in 3×3 matrices [16].

We tried to reduce the number of cycles and the number of contexts for each matrix except an all zero matrix. We fixed the number of contexts at 3 when changing the number of cycles. On the other hand, we fixed the number of cycles at 9 when changing the number of contexts. Note that there is one extra cycle required for the initial condition. Table 3 shows the results. The minimum possible number of cycles was reduced in proportion to the number of zeros. Only when the number of zeros was 3 or 6, the number of required cycles changed depending on the places of zeros. The number of required contexts reduced by one for one matrix when the number of zeros was 5 or 6. For lager number of zeros, the minimum possible number of contexts was 2 when it was 7, and 1 when it was 8.

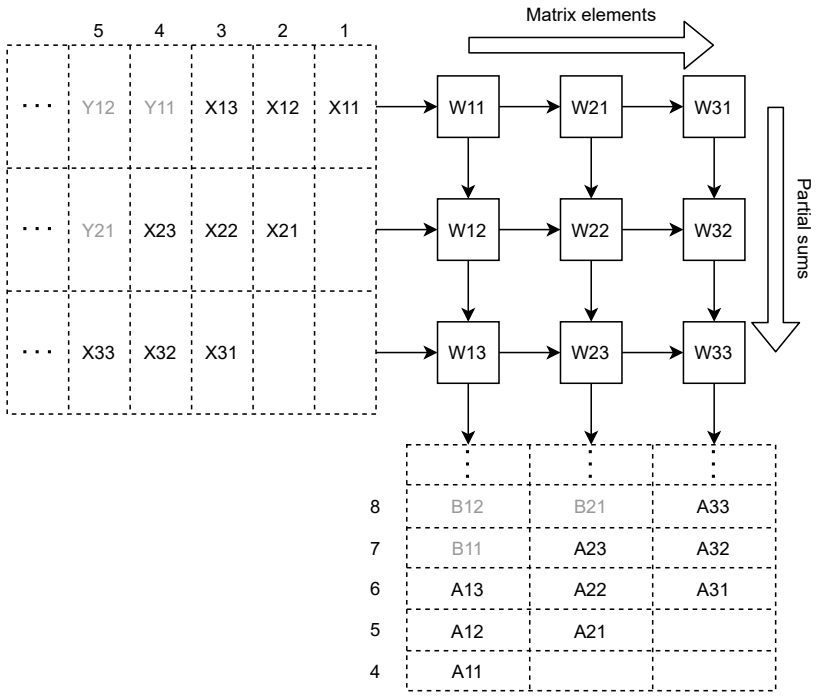

Fig. 6. The algorithm for 3×3 matrix multiplication using nine PEs connected in a 3×3 square mesh.

6 XBTree-based transformation

6.1 CNF formulation

In this work, we implemented another transformation method based on XBTrees, which is originally used for logic factoring of logic circuits [4]. This method can perform the same transformation as the enumeration-based method if the transformation is done for associative and commutative two-input operators.

An XBTree is a binary tree with exchangers that rotate the order of inputs according to the control signals. It can efficiently enumerate all possible structures of binary trees with a specific number of leaf nodes. For an exchanger of size S, let $x_0, ..., x_{S-1}$ and $y_0, ..., y_{S-1}$ be the inputs and outputs respectively. The control signal of the exchanger, c, is an integer in the range from 0 to $S-1$. The function of the exchanger is shown at (4). For example, an XBTree with four leaf nodes is shown at Fig. 7. Depending on the control signal of the exchanger, it is either Fig. 1 (a) or (b), assuming the internal nodes are operator-nodes of addition.

$$\forall n.y_n = x_{n+c \bmod S} \tag{4}$$

Table 3. The number of 3×3 sparse matrices successfully mapped onto the CGRA of nine PEs in a 3×3 square mesh for each number of cycles (when the number of contexts was 3) and for each number of contexts (when the number of cycles was 9).

Zero	Cycle						Context		
	4	5	6	7	8	9	1	2	3
0	0	0	0	0	0	1	0	0	1
1	0	0	0	0	1	1	0	0	1
2	0	0	0	0	3	3	0	0	3
3	0	0	0	1	6	6	0	0	6
4	0	0	0	7	7	7	0	0	7
5	0	0	0	7	7	7	0	1	7
6	0	0	1	6	6	6	0	1	6
7	0	0	3	3	3	3	0	3	3
8	0	1	1	1	1	1	1	1	1

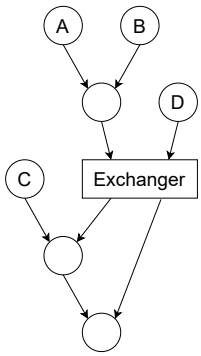

Fig. 7. An XBTree with four leaf nodes: A, B, C, and D.

We can apply the XBTree-based transformation to two-input operators that are both associative and commutative. We create cluster-nodes as described in the previous section and replace each cluster-node by an XBTree and a sorter instead of enumerating all possible orders of operations. For example, a cluster node of multiplication of four variables is replaced by the data-flow graph shown at Fig. 8. The XBTree has as many leaf nodes as the number of inputs to the cluster-node, while the leaf nodes are the outputs of the sorter that reorders the inputs to the order designated by its control signal. The sorter is required to fully search the variants under the commutative law. We implement a sorter as a set of one-output multiplexers, where each multiplexer exclusively selects one from the inputs of the sorter.

The CNF formula needs to be modified to accommodate exchangers and sorters, which are called blocks and not treated as nodes. We introduce new kinds of boolean variables shown below. Let p, q, and r be integers. We number the blocks in a data-flow graph and denote the p-th block by block p and its size (the number of its inputs) by S_p.

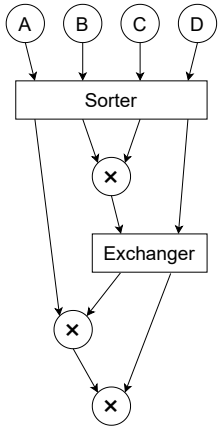

Fig. 8. A data-flow graph implicitly enumerating all possible orders of operations for multiplication of four variables: A, B, C, and D.

- $P_{p,q,j,k}$ ... the q-th output of block p exists in component j at cycle k ($q \in [0, S_p - 1]$)
- $P_{p,q,r,j,k}$ ... the q-th output is the r-th input in block p, and the r-th input of block p exists in component j at cycle k ($q, r \in [0, S_p - 1]$)
- $Q_{p,q}$ or $Q_{p,q,r}$ ... control signal of block p ($q, r \in [0, S_p - 1]$)

We adopt the one-hot encoding for control signals. The variables $\{Q_{p,q}$ for $\forall q \in [0, S_p - 1]\}$ are used as a control signal for block p that is an exchanger, and the variables $\{Q_{p,q,r}$ for $\forall q \in [0, S_p - 1]\}$ are used as a control signal for the multiplexer generating the r-th output of block p that is a sorter.

We add the following four types of clauses to the CNF. Let $d_{p,r}$ denote the r-th input of block p. It can be a node or an output of another block. In the latter case, where it is the q'-th output of block p', $d_{p,r}$ is a pair (p', q'). For simplicity, when d is (p', q'), $X_{d,j,k-1}$ is regarded as $P_{p',q',j,k}$, and $Y_{d,h,k}$ is constant-false (excluded from clauses). The same applies for the elements in D_i, a set of operands, used in the clause 6, and the elements in $D_{i,l}$ in the clause 6.1. A one-hot constraint is a combination of an at most 1 constraint and a large clause containing all literals in the set to make at least one of them true.

11. $\neg P_{p,q,r,j,k} \vee Q_{p,r-q \bmod S_p}$ for $\forall (q, r) \in [0, S_p - 1]^2$, for $\forall p$ where block p is an exchanger, for $\forall j$ where component j is a PE, for $\forall k \neq 0$
12. $\neg P_{p,q,r,j,k} \vee Q_{p,q,r}$ for $\forall (q, r) \in [0, S_p - 1]^2$, for $\forall p$ where block p is a sorter, for $\forall j$ where component j is a PE, for $\forall k \neq 0$
13. $\neg P_{p,q,r,j,k} \vee X_{d_{p,r},j,k-1} \vee \bigvee_{h \in H_j} Y_{d_{p,r},h,k}$ for $\forall (q, r) \in [0, S_p - 1]^2$, for $\forall j$ where component j is a PE, for $\forall (p, k \neq 0)$
14. $\neg P_{p,q,j,k} \vee \bigvee_{r \in [0, S_p - 1]} P_{p,q,r,j,k}$ for $\forall q \in [0, S_p - 1]$, for $\forall j$ where component j is a PE, for $\forall (p, k \neq 0)$
15. One-hot constraint on $\{Q_{p,q}$ for $\forall q \in [0, S_p - 1]\}$, for $\forall p$ where block p is an exchanger

16. One-hot constraint on $\{Q_{p,q,r}$ for $\forall q \in [0, S_p - 1]\}$, for $\forall r \in [0, S_p - 1]$, for $\forall p$ where block p is a sorter
17. At most 1 constraint on $\{Q_{p,q,r}$ for $\forall r \in [0, S_p - 1]\}$, for $\forall q \in [0, S_p - 1]$, for $\forall p$ where block p is a sorter

The clauses 11 and 12 make $P_{p,q,r,j,k}$ false when the q-th output is not the r-th input in block p according to the control signal. The clause 13 then ensures that the r-th input of block p is available in component j at cycle k. The clause 14 finally determines the presence of the q-th output of block p in each component at each cycle. Note that we do not care the cases where component j is not a PE because such $P_{p,q,j,k}$ will never be used (especially in the clause 6 or 6.1). The clauses 15 and 16 make the control signals one-hot. The clause 17 prohibits any two multiplexers in a sorter from selecting the same input.

A minor difference from the enumeration-based method is that intermediate values cannot be shared. For example, when we calculate $A+B+C$ and $A+B+D$, it might be good to calculate $A + B$ and use it to calculate both $(A + B) + C$ and $(A + B) + D$. The enumeration-based method can do that by sharing a table among the cluster-nodes, but the XBTree-based method cannot because it creates an XBTree separately for each cluster-node.

The multi-node operation can be supported by using the clauses 6.1 and 6.2 even if we use the XBTree-based method. After inserting XBTrees, we traverse the data-flow graph to find nodes that match a multi-node operation. During this process, we may encounter the places where nodes are separated by blocks but match a multi-node operation if the control signals for the blocks take a particular value. In this case, we create $R_{i,l}$, a set of control signal variables ($Q_{p,q}$ and $Q_{p,q,r}$) that are true when the control signals take that particular value, while adding the set of operands as a new (l-th) candidate to calculate node i. Then, we disable that candidate unless the control signals take that value by adding the following clause. Note that we do not care the control signal variables that are false because the control signals are one-hot encoded.

6.3. $\neg Z_{i,j,k,l} \vee Q$ for $\forall Q \in R_{i,l}$, for $\forall l \in [0, L_i - 1]$ where $R_{i,l}$ exists, for $\forall (i, j)$ where node i is an operator-node and component j is a PE, for $\forall k \neq 0$

6.2 Comparison: matrix-vector multiplication

We solved the same problem as in the preliminary experiment in Section 3 to compare the automatic transformation methods. We enabled MAC operation in this comparison. We also changed the size of the problem (the size of matrix and the number of PEs) to see the scalability of the methods. We used KISSAT.

The results and runtime are shown at Table 4. TO (Timeout) was set at three hours. When the problem size was 4, the minimum possible number of cycles was 8 (two cycles reduced) just by using MAC operation. This number cannot be more than the number of cycles required to map the data-flow graph where each set of addition and multiplication is manually converted into a three-input operator-node of MAC operation. When the automatic transformation under the

Table 4. The results and runtime in seconds for mapping matrix-vector multiplication using MAC operation with or without the automatic transformation under the associative and commutative laws which was performed by the enumeration-based method (Enum) or the XBTree-based method (XBTree).

Size	Node(Block)			Cycle	Result(Runtime)		
	w/o	Enum	XBTree		w/o	Enum	XBTree
4	48(0)	80(0)	48(8)	6	UNSAT(<0.1)	UNSAT(0.2)	UNSAT(0.1)
				7	UNSAT(0.5)	SAT(1.2)	SAT (0.8)
				8	SAT(0.1)	SAT(0.3)	SAT(0.5)
5	75(0)	185(0)	75(12)	7	UNSAT(0.2)	UNSAT(3.7)	UNSAT(0.8)
				8	UNSAT(2.8)	SAT(761.1)	SAT(109.8)
				9	SAT(1.8)	SAT(61.0)	SAT(4.2)
6	108(0)	420(0)	108(16)	8	UNSAT(0.3)	TO(>10800)	UNSAT(2879.8)
				9	UNSAT(45.7)	TO(>10800)	SAT(3284.0)
				10	SAT(54.0)	SAT(4248.2)	SAT(92.8)

associative and commutative laws was done, the minimum possible number of cycles became 7. It spends one cycle for the initial condition, one cycle just loading inputs, another cycle loading inputs and calculating initial products, three cycles loading inputs and performing MAC operations, and one cycle storing the outputs. The number of cycles also reduced by 1 for the problems of size 5 and 6 by the automatic transformation.

Regarding the comparison between the enumeration-based method and the XBTree-based method, some problems ended up in TO in the enumeration-based method when the problem size was 6 probably because of the exponential increase in the number of intermediate-nodes. On the other hand, the XBTree-based method was able to solve those problems and worked faster than the enumeration-based method for most of the other problems.

7 CGRA optimization

We conducted another experiment to optimize an architecture of CGRA with incremental SAT solving [5]. Up to here, we have considered the methods to adapt data-flow graphs to CGRAs, but we can also optimize CGRAs through iterative synthesis. After getting a minimum cycle schedule, we try to reduce the components and paths one by one without increasing the number of cycles. In this process, we utilize incremental SAT solving, which can reuse the clauses added and learned in the previous calls. Specifically, we solve the CNF, where we obtained a minimum cycle schedule, again with the assumptions (a set of literals that are forced to be true) to disable one component or path. If it is SAT, we add those assumptions to the CNF as clauses, then the component or path will never be used in mapping. Otherwise, we give up removing that component or path. We repeat this for each component and path in the CGRA.

We targeted a data-flow graph generated for AES [17] shown at Fig. 9. It consists of 138 nodes where each operator-node corresponds to a subroutine. We

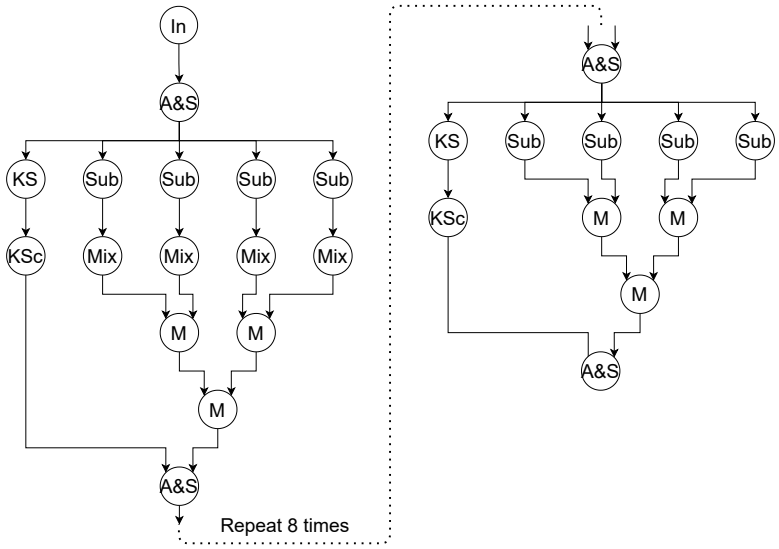

Fig. 9. A data-flow graph for AES.

used a CGRA of 3×3 square mesh PEs shown at Fig. 10. Each PE has one operation-unit and two registers. The mapping was done with no pipelining and no transformation. We used another SAT solver, Glucose v4.1, which supports incremental SAT solving.

The original mapping problem was solved with 52 cycles in 0.5 seconds. It is the theoretical minimum number of cycles because the data-flow graph has 50 levels of operator-nodes and we need one cycle for the initial condition and another cycle for storing the result. Compared to the mapping problem of matrix vector multiplication onto a ring architecture, this problem was solved very fast even though the data-flow graph has more than a hundred nodes. It means that the mapping difficulty comes from not only the number of nodes but also the capacity of the architecture.

Next, we ran incremental SAT solving to optimize the CGRA. We first removed as many PEs as possible, and then removed as many paths as possible. The result is shown at Fig. 11. The optimization took only 1.8 seconds. It turned out that we can sequentially map the data-flow graph onto four PEs connected in a ring, where some PEs are connected in two-way, but others are in one-way. Note that we checked redundancy of PEs (and paths) in a specific order, and it may be better to explore different orders.

8 Conclusion

We proposed a SAT-based data-flow graph mapping method for CGRAs. It performs the automatic transformation under the associative and commutative

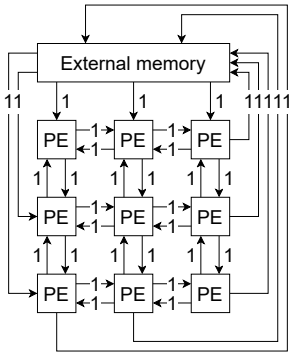

Fig. 10. A CGRA consisting of nine PEs connected in a 3×3 square mesh.

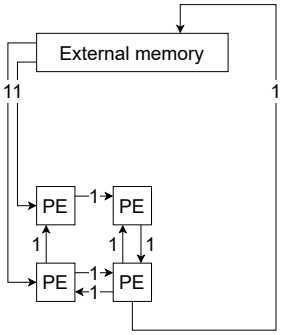

Fig. 11. The CGRA optimized through incremental SAT solving.

laws using XBTrees and sorters. We compared the XBTree-based transformation method with the enumeration-based method, and the XBTree-based method worked faster and solved more problems than the enumeration-based method. In another experiment, we optimized an architecture of CGRA through incremental SAT solving.

Regarding the ILP solver, it was slower than the SAT solver probably because the mapping problem contained few at most K constraints and K was small. If K is large (each PE has a large number of registers for example), the ILP solver might work faster than the SAT solver. Also, we used incremental SAT solving for optimization, but one can use the ILP solver instead.

Our method using SAT solver is not as scalable as the heuristic methods like simulated annealing. We are considering decomposing a data-flow graph or imposing some heuristic constraints by generalizing small mapping results. We are currently working on a hierarchical mapping method, which maps nodes while partitioning the array.

We are also considering adopting a rule base transformation, where a rule is a possible transformation defined by a user, to deal with other than the associa-

tive and commutative laws. For CGRA architecture optimization, it might be good to further explore the search space: the topology of CGRA, the number of operation-units, the number of registers, and the bandwidth of paths.

The source code of our program is available at [18].

References

1. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet Classification with Deep Convolutional Neural Networks. In: Proceedings of International Conference on Neural Information Processing Systems, pp. 1097–1105 (2012)
2. Jouppi, N.P., et al.: In-Datacenter Performance Analysis of a Tensor Processing Unit. ACM SIGARCH Computer Architecture News **45**(2), 1–12 (2017). DOI 10.1145/3140659.3080246
3. Liu, L., Zhu, J., Li, Z., Lu, Y., Deng, Y., Han, J., Yin, S., Wei, S.: A Survey of Coarse-Grained Reconfigurable Architecture and Design. ACM Computing Surveys (CSUR) **52**(6), 1–39 (2020). DOI 10.1145/3357375
4. Yoshida, H., Fujita, M.: Exact Minimum Factoring of Incompletely Specified Logic Functions via Quantified Boolean Satisfiability. IPSJ Transactions on System LSI Design Methodology **4**, 70–79 (2011). DOI 10.2197/ipsjtsldm.4.70
5. Audemard, G., Lagniez, J.M., Simon, L.: Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 7962 LNCS, pp. 309–317. Springer, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-39071-5_23
6. Mei, B., Vernalde, S., Verkest, D., De Man, H., Lauwereins, R.: Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. IEE Proceedings - Computers and Digital Techniques **150**(5), 255 (2003). DOI 10.1049/ip-cdt:20030833
7. Chin, S.A., Anderson, J.H.: An architecture-agnostic integer linear programming approach to CGRA mapping. In: Proceedings of Design Automation Conference (DAC), pp. 1–6 (2018). DOI 10.1145/3195970.3195986
8. Greene, J.W.: Exact mapping of rewritten linear functions to configurable logic. In: Proceedings of International Workshop on FPGAs for Software Programmers (FSP), pp. 11–18 (2019)
9. Chin, S.A., Sakamoto, N., Rui, A., Zhao, J., Kim, J.H., Hara-Azumi, Y., Anderson, J.: CGRA-ME: A unified framework for CGRA modelling and exploration. In: Proceedings of International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 184–189 (2017). DOI 10.1109/ASAP.2017.7995277
10. Flynn, M.J., Pell, O., Mencer, O.: Dataflow supercomputing. In: Proceedings of International Conference on Field Programmable Logic and Applications (FPL), pp. 1–3 (2012). DOI 10.1109/FPL.2012.6339170
11. Miyasaka, Y., Fujita, M.: Sat-based mapping of data-flow onto array processor. In: 2020 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC) (2020)
12. Nguyen, V.H., Mai, S.T.: A New Method to Encode the At-Most-One Constraint into SAT. In: Proceedings of International Symposium on Information and Communication Technology (SoICT), vol. 03-04-Dece, pp. 1–8 (2015). DOI 10.1145/2833258.2833293

13. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 3709 LNCS, pp. 827–831. Springer, Berlin, Heidelberg (2005). DOI 10.1007/11564751_73
14. Lee, G., Choi, K., Dutt, N.D.: Mapping Multi-Domain Applications Onto Coarse-Grained Reconfigurable Architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **30**(5), 637–650 (2011). DOI 10.1109/TCAD.2010.2098571
15. Yoon, J., Shrivastava, A., Sanghyun Park, Minwook Ahn, Yunheung Paek: A Graph Drawing Based Spatial Mapping Algorithm for Coarse-Grained Reconfigurable Architectures. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **17**(11), 1565–1578 (2009). DOI 10.1109/TVLSI.2008.2001746
16. Živković, M.: Classification of small (0,1) matrices. Linear Algebra and its Applications **414**(1), 310–346 (2006). DOI 10.1016/j.laa.2005.10.010
17. Liu, B., Baas, B.M.: Parallel AES Encryption Engines for Many-Core Processor Arrays. IEEE Transactions on Computers **62**(3), 536–547 (2013). DOI 10.1109/TC.2011.251
18. URL https://github.com/MyskYko/dfgmap