Loop Optimization Based on Reinforcement Learning Using the Polyhedral Model

Hayato Takahashi

Graduate School of Science and Technology
Kumamoto University
kumamoto, JAPAN, 860-8555
h_takahashi@st.cs.kumamoto-u.ac.jp

Motoki Amagasaki

Faculty of Advanced Science and Technology
Kumamoto University
kumamoto, JAPAN, 860-8555
amagasaki@cs.kumamoto-u.ac.jp

Masato Kiyama

Faculty of Advanced Science and Technology
Kumamoto University
kumamoto, JAPAN, 860-8555
masato@cs.kumamoto-u.ac.jp

Kenshu Seto

Research and Education Institute
for Semiconductors and Informatics (REISI)
Kumamoto University
kumamoto, JAPAN, 860-8555
k-seto@cs.kumamoto-u.ac.jp

Mery Diana

Research and Education Institute
for Semiconductors and Informatics (REISI)
Kumamoto University
kumamoto, JAPAN, 860-8555
merydiana@kumamoto-u.ac.jp

Abstract— In high-level synthesis technology, optimizing code especially nested loops is often required in order to improve the performance of the generated hardware. While the polyhedral model facilitates the construction of a complete nested loop, finding an optimal solution minimizing instruction executions remains challenging. This paper proposes using reinforcement learning to effectively explore the optimal loop structure. Performance evaluation is conducted by analyzing the number of instructions executed in the generated code and the circuit synthesized by the high-level synthesis tool.

I. Introduction

High-level synthesis technology automatically generates hardware description language from high-level languages such as C, but it is necessary to optimize the code before high-level synthesis in order to maintain the performance of the automatically generated hardware. In particular, when there are multiple nested loops, each must be executed sequentially, which requires a huge number of executions. In this study, we focus on loop fusion, which is used for loop optimization.

Loop fusion is an optimization technique that merges multiple loops into a single loop, which improves performance by reducing execution time, reducing the number of cycles, and increasing data locality. Figure 1 shows an example of loop fusion, with S0 and S1 indicating the executed instruction statements. In loop fusion, multiple loops as in the left of Figure 1 are merged into a single nested loop as in the middle and right. However, in many cases it is difficult to obtain an optimal solution with the fewest cycles as a result of loop fusion. Compared to the middle of Figure 1, the right shows that the number of cycles has been reduced from 200 before the fusion to 100 after, which is because parallel execution is possible in the right of Figure 1.

However, it is not always safe to perform loop fusion. Optimization via loop fusion must be performed under the condition that either there are no dependencies between data before and after fusion or any such dependencies are preserved after fusion. If fusion is performed ignoring data dependencies, then the integrity of the calculation results may be compromised. In this study, we examine an optimization method for this loop fusion based on the polyhedral model using machine learning and particularly reinforcement learning.

Pluto[1] is an automatic parallelization tool that performs high-level transformations such as nest loop op-

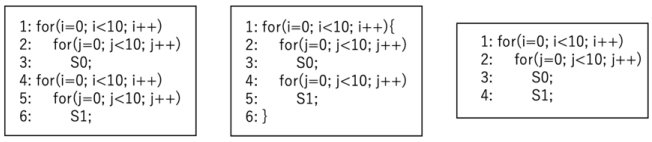

Fig. 1. Example of loop fusion

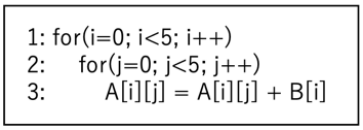

Fig. 2. Example of nested loops

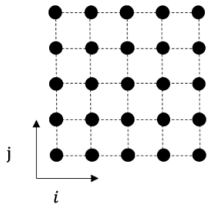

Fig. 3. Integer lattice of points in Figure2

timization and parallelization. It utilizes a polyhedral model for compiler optimization and proposes automatic parallelization technology by finding affine transformations for efficient tiling, thereby increasing parallelism and improving memory reference locality. In OLS[2], performance is improved by performing post-processing on the polyhedral model obtained by Pluto. Specifically, it performs processing to replace dependencies filled with inner loops, which cause performance degradation, with outer loops. From this, it can be seen that the performance of OLS depends on preprocessing such as Pluto, and there are cases where its optimality is limited. In this study, we propose an automatic generation method for fully nested loops by learning this functionality using an optimization technique based on polyhedral models and reinforcement learning, aiming to improve performance.

This paper is organized as follows. Section 2 provides background information, Section 3 explains the proposed method, Section 4 presents the experimental results, and Section 5 concludes.

II. Background

A. Polyhedral Model

This subsection briefly explains the polyhedral model [3], which is a mathematical technique used in compiler optimization. In particular, it is used widely for loop optimization in numerical and scientific computing [4]. The polyhedral model comprises iterative domains and multidimensional schedules, among other aspects. The repetition vector of a statement S is defined as $\vec{i}_S = (i_1, i_2, \ldots, i_{m_S})$. Here, $i_1, i_2, \ldots, i_{m_S}$ is the sequence of loop index variables that contain the instruction statement S, and they are ordered from the outermost loop i_1 to the innermost loop i_{m_S}. The repetition domain D of the instruction statement S is the set of values that $\vec{i}_S$ can take. In the polyhedral model, this repetition domain is expressed as a set of points on a multidimensional integer lattice, and it is interpreted as a polyhedron. For example, the repetition vector of the instruction statement S in the double loop shown in Figure 2 is expressed as $\vec{i}_S = (i, j)$, and this repetition domain can be visualized as the 5×5 lattice of points shown in Figure 3.

An important concept in the polyhedral model is the schedule, which is a mathematical expression that assigns the execution order of instruction statement S and is expressed as a multidimensional vector such as $S_i = (i_0, i_1, \ldots, i_N)$. Each dimension of this vector is composed of constants and loop variables, and it is executed from left to right. The multidimensional vector of the schedule contains a scalar dimension expressed only by constants and a loop dimension expressed by a loop variable. The scalar dimension determines the execution order of the loop, and the loop dimension indicates the loop to be executed. In this study, we make changes to this schedule to convert the execution order and apply various optimizations using the polyhedral model. The optimized polyhedral model is ultimately converted to code in C or another language and output in an executable form.

B. Outer Loop Shifting

OLS[2] is a technique that enables the complete fusion of loops that could not be fused in Pluto. Loop fusion, which fuses multiple loops to generate efficient hardware in high-level synthesis and enables parallel scheduling of operations, contributes significantly to performance improvement. Pluto, an optimization method based on a polyhedral model, is powerful because it can optimize locality and outer parallelism simultaneously, enabling automatic loop fusion. However, in some cases, it cannot completely fuse loops, resulting in performance that does not fully utilize HLS. In response, Kato et al. proposed Outer Loop Shifting (OLS) as a post-processing step for Pluto. OLS is a technique that shifts the scalar dimension in a multidimensional schedule to the outer loop, enabling the complete fusion of loops that could not be fused by Pluto. While Pluto alone achieves only partial fusion in benchmarks, applying OLS enables the complete fusion of all loops and reduces the execution cycle. However, this method is a deterministic approach that seeks the mathematically optimal solution and performs post-processing,

so its performance depends on the pre-processing. OLS uses Pluto as its preprocessing step, which is an optimization technique targeting compilers. While Pluto's improvements in parallelism and locality affect HLS performance, OLS's performance depends on the optimization results of Pluto. In contrast, this method explores flexible search that is not limited to the solutions derived by Pluto by directly manipulating the polyhedral model using reinforcement learning.

C. Reinforcement Learning

Reinforcement learning [5] is a machine-learning method whereby agents learn optimal action policies via repeated trial and error through interaction with a dynamic environment. In this method, the aim of an agent is to choose an action that maximizes its reward. Specifically, the agent receives the current state from the environment as input and decides on an action based on this. The environment receives the agent's action, presents a new state, and gives the agent a reward based on the action. The agent uses this feedback to learn how to select actions that maximize reward. Reinforcement learning provides a framework that can effectively explore vast search spaces through trial and error with randomness. Therefore, compared to deterministic methods that derive solutions based on explicit analysis and designer knowledge, reinforcement learning enables more flexible and diverse optimization.Additionally, reward design enables exploration with high degrees of freedom. For example, by incorporating HLS tools into the environment, metrics such as start interval II, latency, and resource usage—which are difficult to formalize mathematically—can be used as rewards, enabling learning that directly improves HLS performance.In this study, due to the complexity of the processing, learning was not performed with HLS tools included in the environment. However, learning was conducted with the aim of improving parallelism and reducing latency using the total number of execution instructions described in Chapter 4. The reinforcement learning model Proximal Policy Optimization (PPO) [6] used in this study is a type of reinforcement learning algorithm based on policy gradient methods proposed by Schulman et al[7]. In the conventional trust region policy optimization (TRPO) [8], a constraint based on the Kullback–Leibler divergence was introduced to prevent the destabilization of learning due to appreciable updates of the policy. However, TRPO had high computational cost, and implementation complexity was also an issue. By contrast, PPO introduces clipping to appropriately limit the update of the policy, and it is possible to ensure the stability of learning with a simple implementation and low computational cost. In PPO, the following loss function is applied to control the change in the policy so that it does not become excessive:

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t\Big[\min\big(r_t(\theta)A_t, \mathrm{clip}\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)A_t\big)\Big], \tag{1}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}. \tag{2}$$

Here, a_t represents action, s_t represents state, r_t represents the probability ratio with respect to the past policy, A_t represents the advantage function, and ϵ represents the clipping range. In PPO, the update amount is limited when the probability ratio r_t exceeds the range of $1-\epsilon$ or $1+\epsilon$, preventing the destabilization of learning due to excessive policy updates.

D. Loop Interchange

Loop interchange is an optimization technique that changes the order of nested loops. Figure 4 shows an example of conversion for multiple nested loops, with S0 and S1 indicating executable statements with loop variables i and j.

This conversion example shows the case where the inner and outer loops of the multiple loop for S0, which is the executable statement in the third line, are swapped. In this case, the schedule for the executable statement in the third line is converted from S0 = $(i, j, 0)$ to S0 = $(j, i, 0)$.

E. Loop Shifting

Loop shifting is an optimization technique that shifts the repetition of a loop back and forth. Figure 5 shows an example of a transformation for a multiply nested loop.

In this conversion example, the loop variables (i, j) of S0, which is the executable statement in the third line on the left of Figure 5, are loop-shifted by a shift amount of 0 and 1, respectively. The schedule for S0 at this time is converted from S0 = $(i, j, 0)$ to S0 = $(i, j-1, 0)$. In this study, the shift amount for one loop shift is limited to ± 1 or no loop shift, and one of the loop shifts is performed on all loop variables contained in the target statement.

III. Proposed Method

Figure 6 shows the overall structure of the present processing flow. First, the input code with parallel nested loops is converted into a polyhedral model. At this time,

```
1: for(i=0; i<5; i++)
2:     for(j=0; j<5; j++)
3:         S0(i,j)
4:         S1(i,j)
```

```
1: for(i=0; i<5; i++)
2:     for(j=0; j<5; j++)
3:         S0(j, i)
4:         S1(i, j)
```

Fig. 4. Before (left) and after (right) loop interchange

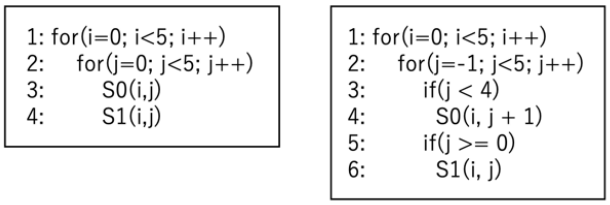

Fig. 5. Before (left) and after (right) loop shift

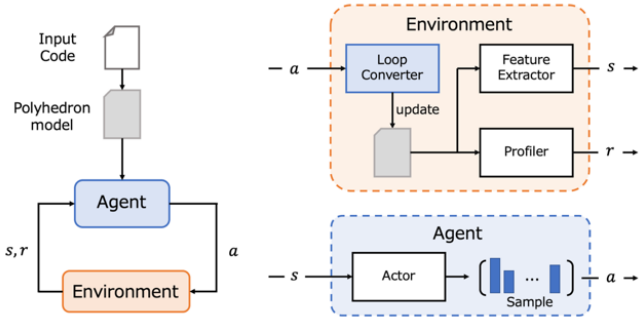

Fig. 6. Processing flow

the parallel nested loops are converted into a single nested loop by changing the schedule of the polyhedral model.

Specifically, first, the number of dimensions in the execution statements in the input program is unified. Here, the dimension with a fixed value in each schedule is deleted, and the schedule is changed to one that only has a loop dimension. After that, the number of dimensions is unified to the schedule with the largest number of dimensions. In addition, by unifying the loop variables in each schedule, it is converted to a single nested loop. Finally, a fixed-value dimension for identification is added to each schedule, and loop fusion is performed. However, because the conversion here does not account for the dependencies of the internal processing but rather formally merges the loops, the single nested loop after the conversion does not satisfy the dependencies between the data, and in many cases it is not optimal. We use PPO, a reinforcement-learning model, to optimize the loop using this incomplete single nested loop as the initial state.

From the state given as input, the agent decides on a loop transformation method for the nested loops and performs schedule transformation. The environment outputs a new state and reward based on the schedule to which the transformation has been applied and passes them to the agent. This series of actions is defined as one step, and loop interchange or loop shift is performed on one execution statement for each step. If we assume that the number of loop variables in the execution statement to which the transformation is applied is n, then there are $n!$ ways to swap the loop variables for loop interchange. For loop shift, since each loop variable is shifted by -1, 0, or 1, there are 3^n possible transformations for a single execution statement. Therefore, the agent's action space A is expressed as follows:

$$A = [a_1, a_2, \cdots, a_i], \tag{3}$$

$$i = \sum_{m=1}^{M} \left(N_m! + 3^{N_m}\right). \tag{4}$$

Here, M represents the number of executable statements contained in the nested loop, and N_m represents the number of loop variables contained in the mth executable statement. The state passed to the agent by the environment uses features obtained from the abstract syntax tree (AST). We use 22-dimensional integer features obtained by analyzing the AST and extracting features of the program. The details are given in Table I. Each feature listed in this table is derived from an analysis of AST and serves as a quantitative measure of the program ' s structural characteristics. Each entry represents the occurrence count of the corresponding component within the program. These numerical values are utilized to assess the impact of loop optimization and are employed as input to the reinforcement learning model.

The reward R is set as a ratio of the total number of execution instructions to the input code. The equation for remuneration is

$$R = -\frac{uniqepoint}{uniqepoint_{original}} \tag{5}$$

where $uniqepoint$ is the total number of execution instructions in the converted code, $uniqepoint_{original}$ is that in the input code.

In this method, the reward is important because the reinforcement learning model is trained to output high-performance loop fusion results. The reward for learning results that will perform well when hardware-implemented is calculated using the number of lattice points in the iteration space after loop transformation. Specifically, it is defined as the number of all lattice points in the iteration space after loop transformation that are executed at least once. Here, we use a figure 7 to explain the total number of executed instructions. This figure shows the iteration space after loop transformation of two execution statements, S0 and S1. This loop is a completely nested loop consisting of an outer loop i and an inner loop j. Therefore, the number of loop executions is 30 in the left figure 7 and 50 in the right figure 7. In these two cases, since the loop is fully nested, the latency can be expressed using the loop count M of loop i, the loop count N of loop j, the start interval II, and the execution cycle L of one loop, as follows.

$$M \times II \times (N-1) \times L \tag{6}$$

TABLE I
22-DIMENSIONAL INTEGER FEATURES

Features	
Number of loops	div instructions
Maximum loop depth	mod instructions
Branch count	and instructions
eq instructions	or instructions
lt instructions	not instructions
le instructions	min instructions
gt instructions	max instructions
add instructions	Function call
sub instructions	Function type count
minus instructions	ge instructions
mul instructions	Basic block count

If L is sufficiently small compared to M and N, the execution cycle can be approximated as.

$$M \times N \times II \tag{7}$$

In a fully nested loop, the range of values that the loop variables can take is limited to the number of loop iterations. Therefore, the fewer the number of grid points in the iteration space after the loop transformation, the fewer the number of the execution cycle. Thus, the total number of execution instructions is used as the reward.

The reward is calculated as the ratio of the total number of execution commands to the input code, and the negative sign is so that the total number of executed instructions in the converted code decreases . However, the actions taken by the agent do not account for the dependency relationships between executable instructions, and there are cases where the converted code cannot be executed. When such illegal actions are selected, the reward is set to -10, and learning is performed so that the agent does not select a conversion that does not satisfy the dependency relationship.

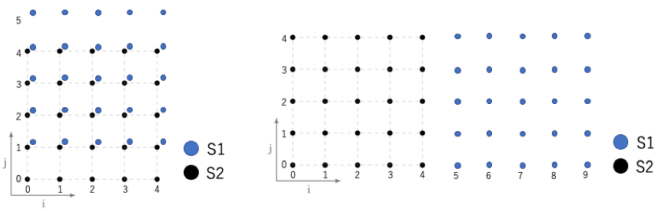

Fig. 7. Lattice points in iteration space

IV. EXPERIMENTAL RESULTS

In this study, we conducted experiments targeting programs that perform matrix calculations involving parallel multiple nested loops. We used Xilinx's high level synthesis tool Vitis HLS 2020.2, with the target device being xcvullp-flga2577-1-e. Python 3.11.10 was used for development and scheduling implementation, with the libraries Islpy, gymnasium, and stable baseline3 being utilized.The reinforcement learning agent performs a single loop transformation decision for the scheduling target loop and receives the state and reward from the environment based on the result. This sequence of operations is defined as one step, and the sequence of steps from the initial state to the final state is defined as one episode. Exploration is performed by restarting the loop transformation decision from the initial state for each episode. In this study, we set 20 steps as the end state and perform a total of 6,400 episodes of learning.

In the proposed method, we fixed the number of episodes to 20 steps and trained the program for 6400 episodes. Figure 8 shows the code before optimization and that after optimization using the proposed method. The code after optimization converts multiple nested loops into a single nested loop. Even after conversion, the dependency between executable instructions is satisfied, so it is an executable code. Figure 9 also shows the average reward obtained by the proposed method over 10 trials. As can be seen, the agent is learning to reduce the total number of instructions executed, as the reward obtained increases as the episode progresses.

Table II gives the results of high-level synthesis for the code before optimization, the code optimized using the proposed method and the code optimized using OLS. As can be seen, the total number of instructions executed has decreased by 50% compared to before optimization, and latency has decreased by 46.61%. This is because parallel execution has become possible by merging loops. However, when compared to the optimization results obtained using OLS, the total number of instructions executed is the same, yet latency has increased, indicating a perfor-

```
for (int i = 0; i < 8; i++) {
  for (int j = 0; j < 8; j++) {
    T[i][j] = 0.0;
    for (int k = 0; k < 8; ++k) {
      T[i][j] = T[i][j] + alpha * A[i][k] * B[k][j];
    }
  }
}

for (int i = 0; i < 8; i++) {
  for (int j = 0; j < 8; j++) {
    D[i][j] = beta * D[i][j];
    for (int k = 0; k < 8; ++k) {
      D[i][j] = D[i][j] + T[i][k] * C[k][j];
    }
  }
}
```

```
for (int i = 0; i <= 8; i += 1) {
  for (int j = 0; j <= 7; j += 1) {
    for (int k = 0; k <= 7; k += 1) {
      if (i <= 7 && k == 0)
        T[i][j] = 0.0;
      if (i <= 7) {
        T[i][j] += alpha * A[i][k] * B[k][j];
        if (k == 0) {
          D[i][j] *= beta;
        }
      }
      if (i >= 1) {
        D[i - 1][j] += T[i - 1][k] * C[k][j];
      }
    }
  }
}
```

Fig. 8. Before (left) and after (right) optimization

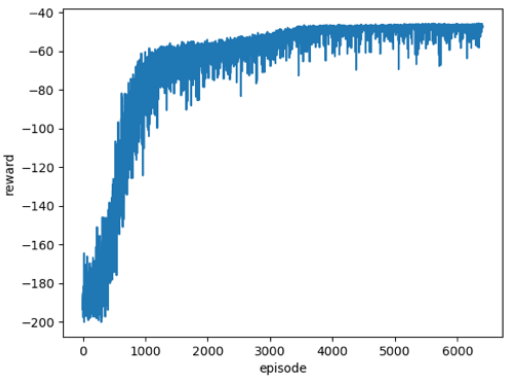

Fig. 9. 10-time average of agent's reward

TABLE II
HIGH-LEVEL SYNTHESIS RESULTS

	No fusion	Optimized	OLS
Total number of execution instructions	1152	576	576
Operating cycle [ns]	10.0	10.0	10.0
Latency [ns]	1697	906	709
DSP	5	13	29
FF	542	3011	3394
LUT	831	2582	3105

mance degradation. One possible factor contributing to this is the reward setting. In this study, the reward was set based on the total number of instructions executed, and the system was trained to reduce this value, aiming to improve parallelism, i.e., reduce the number of execution cycles. However, optimization targeting locality was not performed.In contrast, OLS performs localization optimization using Pluto, which is thought to have reduced memory access and led to shorter latency. Therefore, it is necessary to perform learning that considers localization in the future. Specifically, we will aim to further reduce latency by including rewards that minimize the dependency distance of the outer loop.

V. CONCLUSION

In this study, we described a method for converting a program into a single nested loop by performing a transformation based on the polyhedral model using reinforcement learning. With this method, we were able to reduce the total number of instructions executed has decreased by 50% and latency by approximately 46% by performing loop fusion on a program with multiple nested loops. However, when compared to OLS, while the total number of instructions executed remained the same, latency increased. This is attributed to the fact that reward settings did not account for locality. Going forward, we will investigate learning that incorporates locality and optimize other programs for further research.

REFERENCES

[1] Bondhugula, Uday and Hartono, Albert and Ramanujam, J and Sadayappan, P, "Pluto: A practical and fully automatic polyhedral program optimization system", *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008). Citeseer*, vol.146, 2008

[2] Yuta Kato and Kenshu Seto, "Loop fusion with outer loop shifting for high-level synthesis.", *IPSJ Transactions on System and LSI Design Methodology*, vol.6, pp.71–75, 2013.

[3] Magnus J. Wenninger, "Polyhedron models.", Cambridge University Press, 1971.

[4] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam and Ponnuswamy Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer.", *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp.101-113, 2008.

[5] Leslie Pack Kaelbling, Michael L. Littman and Andrew W. Moore, "Reinforcement learning: A survey", *Journal of Artificial Intelligence Research*, vol.4, pp 237-285, 1996.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov, "Proximal policy optimization algorithms", arXiv preprint arXiv:1707.06347, 2017.

[7] Jan Peters and Stefan Schaal, "Policy gradient methods for robotics.", *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp.2219-2225, 2006.

[8] John Schulman, "Trust Region Policy Optimization.", arXiv preprint arXiv:1502.05477, 2015.