Neurocomputing 481 (2022) 102–120

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Deep reinforcement learning in loop fusion problem

Mahsa Ziraksima, Shahriar Lotfi *, Jafar Razmara

Department of Computer Science, Faculty of Mathematics, Statistics and Computer Science, University of Tabriz, Tabriz, Iran

ARTICLE INFO

Article history:
Received 12 May 2021
Revised 7 December 2021
Accepted 14 January 2022
Available online 21 January 2022
Communicated by Zidong Wang

Keywords:
Parallelizing compilers
Loop optimization
Loop fusion
Data dependence
Deep reinforcement learning

ABSTRACT

Loops' execution time and resource consumption are one of the interest points and vital issues in the field of appraising complex scientific or computational algorithms. This issue caused the proposal of Loop performance optimization techniques such as fusion. In the literature, loop fusion merges the loops by taking into account a set of properties associated with the loops or the system on which the resulting code will be executed. The number of these factors and their interactions on the one hand, and the high runtime of available comprehensive approaches, on the other hand, reveals the need for a new method that could be concerned for further progress in solving this NP-hard problem. For the first time, *Deep Reinforcement Learning Loop Fusion* (DRLLF) advanced to be an ideal solution for the challenge in this article. For the proposed framework, a particular matrix is configured as the inputs of a deep neural network based on the information of the problem, namely data dependencies, data reuse, loops' types, and computer system's register size. These randomly generated matrixes are used in the training phase by reinforcement learning to get the imperative experience on predicting a profitable distribution over loops' various fusion orders. In the evaluations performed, the presented algorithm was able to achieve the same or better performance in terms of speedup rate, comparing with the methods under study, approximately averaged in 7.36 percent better results. The considerable improvement observed in the results, besides the low run time, proves the comprehensiveness and superiority of this approach.

1. Introduction

Formerly in parallelization, a programming team would first implement the sequential code of a program, and then, an expert team would convert it to the equivalent parallel code. Later on, as computer architecture became more and more complex according to Moore's Law, it became strenuous and time-consuming to obtain the proper performance. At this point, the use of parallelizing compiler was introduced, which automatically detects points in sequential code that have the potential for parallelization and converts them into equivalent parallel versions [1,2]. Another way to say, parallelizing refers to dividing some parts of the program that can run simultaneously into several subroutines and run them in parallel with the aim of achieving superior speed.

With the advancement of technology and computer science, the growing gap between processors and memory speeds has made program execution speed more concerned with memory management [3,4]. Researchers have addressed the issue of reducing memory latency in two fields of loop transformation and data transformation [5]. Loop is a controlling instruction in programming whose operations are repeated several times depending on its conditions as loop index, loop bound, and its steps, forming the iteration space of a loop. Due to the loops' leading part in computer systems' resource consumption and their dominant impact on performance, they are one of the most significant components of a program and the best candidates for parallelization. Deferent types of loop transformation can improve performance, runtime, and power consumption by creating an equivalent program in which data reuse and the number of memory accesses are enhanced [6,7].

On the way to reach this goal, some problems arise that are NP-complete or undecidable [8]. Transformations such as distribution [6,9] or fusion [10] are some cases in point. Loop fusion is a loop transformation in which several loops are combined to form a loop. In contrast, loop distribution is a transformation that breaks the body of a large loop into several smaller ones.

Locality improvement has also been studied extensively, and methods such as blocking [11], unroll-and-jam [12,13], and loop tiling [14,15,16] have been proposed. Amid these transformations, the improvement of data reuse has received less attention, but fusion focuses on this point by considering the correlation between

* Corresponding author.
E-mail addresses: m.ziraksima@tabrizu.ac.ir (M. Ziraksima), shahriar_lotfi@tabrizu.ac.ir (S. Lotfi), razmara@tabrizu.ac.ir (J. Razmara).

https://doi.org/10.1016/j.neucom.2022.01.032
0925-2312/

loops and increasing the likelihood of finding the data in register or cache [17,18].

In a parallel system, the execution of two loops, as in Fig. 1, is accompanied by a synchronization between them. However, fusion decreases the number of required synchronizations by merging loops and thus increasing their granule size [1,19]. It is also present at the instruction level [20]. Since synchronization is a costly operation and necessary after each parallel loop, minimizing its count is very important [19,21]. In this way, not only fusion reduces the loop overhead but also increases the possibility of data reuse [22,10].

It is necessary to point out that the fusion's objectives are not always along the same lines. For example, fusing some loops may cause a dependence based on loop iterations, resulting in the loss of parallelism, or more precisely fine-grained parallelism. On the other hand, over-increasing the code size may boost the register pressure and elaborate the program control flow. Thereby, it could downturn program execution speed by declining memory and subsequent optimizations performance [23,24]. Apart from that, various types of transformations in optimization compilers, involving fusion, can affect each other's results. Consequently, some scholars are privileged with examining a set of these transformations in order to achieve better results by taking their advantage collectively [25–29].

This paper tackles the loop fusion problem by proposing a deep reinforcement learning method. Accordingly, it introduced an algorithm with the subsequent key contributions:

- Finding a proper order for loop fusion
- Enhancing parallelism
- Considering the system's register size as a boundary for fusion
- For the first time, commencing to use a robust framework as deep reinforcement learning to properly solve loop fusion problems in a short span of time.

Hereafter, various fusion studies are reviewed in the next section. The definition of the fusion problem, its fundamental conditions, and challenges are discussed more precisely in section 3. Also, it addresses the proposed deep reinforcement learning method. Then, section 4 describes in detail the implementation of the proposed method's distinctive components. Section 5 evaluates the results, followed by section 6, which assesses its merits and demerits. Section 7 concludes this scrutiny's findings and suggests further potential research questions.

2. Related work

In literature, there are numerous studies on the loop fusion problem, including Warren's research in 1984 [30], which introduced a new structure called hierarchical dependence graph (HDG) as an intermediate representation of codes that allows large parts of the program to be abstracted well. He viewed fusion problem modeling via this graph as a case in point. It can be said that they came up with an acceptable template, but unfortunately, they did not provide any implementation or develop their work on other transformations.

Fig. 1. A synchronization requisite between loops.

Kennedy and McKinley [31] raised two problems in loop fusion and offered different solutions for each. The first problem is fusing a set of parallel and sequential loops with the aim of minimizing the synchronization cost without losing their parallelism. To do this, subgraphs of parallel and sequential nodes and the edges between them are created separately, and the fusion is applied in Breadth-first order. The second problem was fusing a set of parallel and sequential loops to improve the locality. In this affair, after partitioning the graph to fusible loops, conforming with the previous algorithm, it traverses the resulting graph from bottom to top and moves some nodes between the segments. The most crucial objective of this research is to examine these issues separately. In [32], they presented a different solution for ordered typed fusion problem. A definite sequence of types determines the fusion priority between different types on the Breadth-first loop order. One impediment of this practice is its demand to delineate the types' priority order.

Singhai and McKinley submitted an ideal solution for the fusion problem based on dynamic programming [33]. This method is applied to a maximum spanning tree associated with the dependence graph and fuses the loops based on the system's register size. The main shortcoming of this greedy algorithm is its application on the Maximum Spanning Tree, owing to removed edges in this tree which may be related to the optimal solution. Megiddo and Sarkar [34] have presented a formulation based on Integer Programming, which tries to derive a permissible grouping of the loops, so that it could remove some edges. The downside of this procedure is its demand for a predetermined order to solve the problem. Darte in [8] examines the loop fusion problem categories in terms of complexity and reveals how a simple graph traversing can resolve the problem's polynomial cases. Fraboulet et al. [35] use fusion as a tool to mitigate memory space consumption by reducing the temporary arrays utilization in embedded systems.

In [36], Kennedy modeled fusion problems as a graph in which the edges are weighted with the number of memory operators that will be lowered by merging them in comparison to the original program. Howbeit, its solutions may not be superlative because of giving priority to fusing nodes with high weighted edges between. Verdoolaege et al. [37] used the distance vector instead of dependence. In this situation, dependencies with the slightest distance-vector take precedence. According to the presented results, its refinement is insignificant because only localization is considered. Ding and Kennedy [38] improved data reuse with a two-step algorithm. In the first step, all the calculations that use the same data are merged with the help of other transformations, and in the second step, the data used in a loop are clustered based on the cache space so that some arrays are merged, and some others are divided. This method also did not properly function in some cases. Marchal et al. [39] suggested that distinct policies could work better in diverse situations. Therefore, three problem-solving policies have been adopted for the most straightforward scheme, the fastest mechanism, and the least amount of energy consumption. The disadvantage of this method may be its different solution for each of these goals, rather than a comprehensive approach to improving all three. In [9], Liu et al. reveal fusion improvement by combining it with loop distribution, but they focused on two-level loops. Qiu et al. [40] could allow more loop fusion via eliminating fusion preventing dependencies via a time reset method that relocates a calculation for a specific iteration number. It also merges nodes in topological order, which may lead to a suboptimal result. Tian et al. [41] proposed a fusion algorithm to improve Stream Register File (SRF) performance in three cases that appeared by taking into account the program transfer time and the data transfer time. The trouble with this method is no considering types for the loops and thus losing parallelism in some cases. Mehta et al. [42] studied the fusion problem in the context

of polyhedral compilers. However, they have not considered criteria such as register pressure.

Most of the recent researches on fusion has focused on its applications [26,27,43–45], not on offering a new comprehensive method to solve it. Acharya et al. [19] introduced a new problem-modeling structure in polyhedral compilers that, by applying a greedy polynomial clustering heuristic, segments the graph via coloring it. In order to maintain parallelism, fusion has been applied according to the method mentioned in [31]. It should be noted that other transformations have also been considered along in the mentioned study. Ziraksima et al. [46] introduced an evolutionary algorithm to acquire an appropriate loop order for fusion by considering the three influencing factors as data reuse, the number of required synchronizations, and systems' register size. Thence, they have good results for different programs and architectures. Contrastingly, fusing imperfect or multi-level loops is not reflected. Moreover, owing to evolutionary algorithms' high execution time, this method may be reasonable when the response time is not prominent [47].

In conclusion, on account of the loop fusion issue's NP-hard nature, diverse methods have been suggested, the primary purpose of which is minimizing execution time with the help of merging loops. To make reviewing them easier, Table 1 summarizes their specifications. As demonstrated, these methods could not accurately provide the features of a comprehensive algorithm for the loop fusion problem. For example, except for Articles [33] and [46], neither of them met all three improvement factors. However, [33] cannot support more than two types of loops. In addition, adjusting the register size based on the number of loops is not acceptable, and it is better to examine the number of arrays instead. As well, the method suggested in [46] requires a lot of time to solve the problem due to the use of the evolutionary approach, which is not always admissible. These points indicate that more research is still imperative in this area. As the first step in this direction, a novel approach is presented based on deep reinforcement learning, the relevant features listed in the last row of Table 1.

3. Material and methods

The fusion problem can be modeled in different ways [8], comprising the Shortest Common Supersequence problem [48,49], Scheduling problem [50,51], Traveling Salesman problem [52,53]. Besides, various conditions of this problem can affect its complexity. Fusion problem with one type of loops, or two types of loops without fusion preventing edge, also ordered typed fusion problem can be resolved in polynomial form. The problem of maximum fusion with fusion preventing edge, for fixed and larger or equal to two number of loop types, also the problem of finding the maximum amount of fusion without fusion preventing edge, for constant and larger or equal to three number of loop types are NP-complete [8]. Moreover, fusing loops with the aim of improving data reuse is NP-hard [31,42].

As pointed out, one of the factors causing this complexity is the multiplicity of loops' types. Loops can be grouped according to different possible diversities between them:

- They are perfect or imperfect; Nested loops are perfect if all program instructions are in the innermost loop. Otherwise, if there are instructions between the loops, they are called imperfect.
- They can be different in terms of upper or lower bound or their steps.
- The number of their levels can be different.
- Also, their execution type can be parallel or sequential.

Another influential factor in the complexity of the loop fusion problem is data dependence which is a constraint between program instructions that ascertain the order in which the instructions should be executed. There is a data dependence between program instructions if they have an executable path between them, both access to one memory location, and at least one of which is storage [54]. Based on the order of reading or writing in memory, there could be three types of dependencies between program instructions, namely true-dependence, anti-dependence, output-dependence. Sometimes the value of a variable is read from memory by two instructions, which is called input-dependence.

Consider the program in Fig. 2, having five loops with disparate data dependencies between them. Regarding the subjects raised in the previous paragraph, there are true, anti, output, input, and fusion preventing dependencies based on array A, C, H, G, and A between the appointed loops, respectively. Additionally, if each loop is indicated by a circle and each dependence with an edge between them, a data dependence graph is obtained, which is shown on the left side of this figure. It should be underlined that different types of loops can be represented by nodes with distinct shapes in the dependence graph.

Dependencies may occur at different loop iterations. If a dependence occurs in a fix iteration, it is called loop independent dependence, and if they occur between various iterations, they are called loop carried dependence. These two types of dependencies determine the possibility of running the loops in parallel [55,56].

For example, since all loops in Fig. 2 have no loop carried dependence, they all are parallel loops. A fusion preventing edge is a type of data dependence that prevents a pair of loops from merging because their fusion will create a loop carrier dependence, in view of the fact that the loop resulting from fusing loop 1 and 5 will have a loop carrier dependence based on array A. In this case, to make sure that the output of the fused program is equal to the original program's output, the resulting loop must be executed sequentially. In other words, the type of the parallel loop has changed to sequential.

In summarizing the issues raised, an algorithm presented to solve this problem must meet the following conditions and criteria to obtain a correct result:

- To preserve the desired output of the original program, the order of the two loops, which are determined by their dependencies, is not mutable.
- If there is a fusion preventing edge between two loops, they are not fusible.
- Two loops with different types, having different bound or one of them being parallel and the other one sequential, cannot be merged even if it does not violate any dependence.
- The dependence graph resulting from the fusion must be acyclic to ensure a definite order between the merged loops.

Given the conditions, the challenge is to put forward a scheme that covers all the objectives. More precisely, fusion should be applied according to the existing dependencies so that a program with the least number of loops is acquired. On the other hand, to improve the actual performance of the resulting program, additional criteria must also be considered, such as advancing parallelism and data reuse, optimizing the number of vital synchronizations and the cost of examining loop bounds, along with some features affiliated to the computer systems' architecture as register pressure. Considering the NP-hardness of the fusion problem, the majority of the researchers in this field have tried to solve it by greedy and heuristic approaches or by applying hypotheses to reduce the complexity of the problem, allowing to handle it by deterministic methods. State-of-the-art and influential optimization frameworks are rarely used in this area. The evolu-

Table 1
Qualitative comparison of DRLLF with comparative works.

[Ref.]	Year	Paper's goal	Number of types	Graph's node	Loop order	Modeling method	Improvement factors		
							parallelism	data reuse	Register size
[30]	1984	Improving parallelism	2	Loop and instruction	Topological order of loop independent dependence graph	hierarchical dependence graph (HDG)	✓	✓	×
[31]	1993	Provide two distinct methods for improving parallelism and data reuse	2	loop	Breath-first-search with the parallel type's priority	Directed acyclic graph	✓	✓	×
[32]	1994	minimizing the number of loops	>2	loop	Breath-first-search with the determined type's priority	Directed weighted acyclic graph	✓	✓	×
[33]	1997	Improving parallelism and data reuse considering register size	2	loop	Bottom-up order	weighted tree	✓	✓	✓
[34]	1997	Improving data reuse with maintaining parallelism	2	loop	Topological order	Directed weighted acyclic graph	✓	✓	×
[8]	2000	Improving parallelism and data reuse	2	loop	Topological order	Directed acyclic graph	✓	✓	×
[35]	2001	Minimizing the use of temporary arrays	1	loop	Topological order	Directed acyclic graph	×	✓	×
[36]	2001	Improving data reuse with maintaining parallelism	>2	loop	The order of selecting the edge with the maximum weight	Directed weighted acyclic graph with undirected input dependence	✓	✓	×
[37]	2003	Improving data reuse	>2	code	Topological order with the priority of instructions that use the same data	Works on code	×	✓	×
[38]	2004	Improving data reuse	>2	instruction	An order with the priority of true-dependence that has minimum distance vector and maximum number of data	Directed graph that is not limited to perfect and imperfect loops	×	✓	✓
[39]	2004	Improving data reuse considering register size	2	loop	Topological order with the priority of removing the edges with maximum weight	Directed weighted acyclic graph	×	✓	✓
[9]	2005	Improving data reuse with considering code size	2	loop	Topological order	Directed acyclic graph	×	✓	✓
[40]	2008	Reducing energy consumption by fusion	>2	loop	Topological order	Multidimensional loop dependence graph (weighted, directional)	✓	✓	×
[41]	2012	Improving data reuse considering register size	1	loop	Based on the sorting algorithm presented in the article	Directed acyclic graph	×	✓	✓
[42]	2014	Improving data reuse with maintaining parallelism	>2	instruction	The presence sequence of nodes in the program, taking into account the strongly connected components	Directed acyclic graph	✓	✓	×
[27]	2016	Fusion of parallel array operations	–	partitions	Topological order obtained by branch-and-bound search	Partition graph	×	×	×
[44]	2017	Loop fusion for program verification	1	Loops and variables	Program order	Works on code	×	×	×
[26]	2018	Fusion and tile size model for optimizing image processing pipelines	–	stages	The order obtained by a specific clustering algorithm	Pipeline graph	×	×	×
[43]	2018	Improving image processing by loop fusion	2	loop	Topological order	Works on code considering data dependences	×	×	×
[45]	2019	kernel fusion	–	kernels	The order obtained by graph partitioning technique with domain-specific and architecture knowledge	Directed acyclic graph	×	×	×
[19]	2020	Improving data reuse with maintaining parallelism	2	permutation	Topological order considering a coloring heuristic	Fusion conflict graph (FCG)	✓	✓	×
[46]	2020	Improving parallelism and data reuse considering register size	>2	loop	The sequence generated by evolutionary algorithm	Directed acyclic graph	✓	✓	✓
Proposed method (DRLLF)		Improving parallelism considering register size with the ability to solve the problem in the shortest possible time	>2	loop	The sequence generated by Deep reinforcement learning	Directed weighted acyclic graph	✓	✓	✓

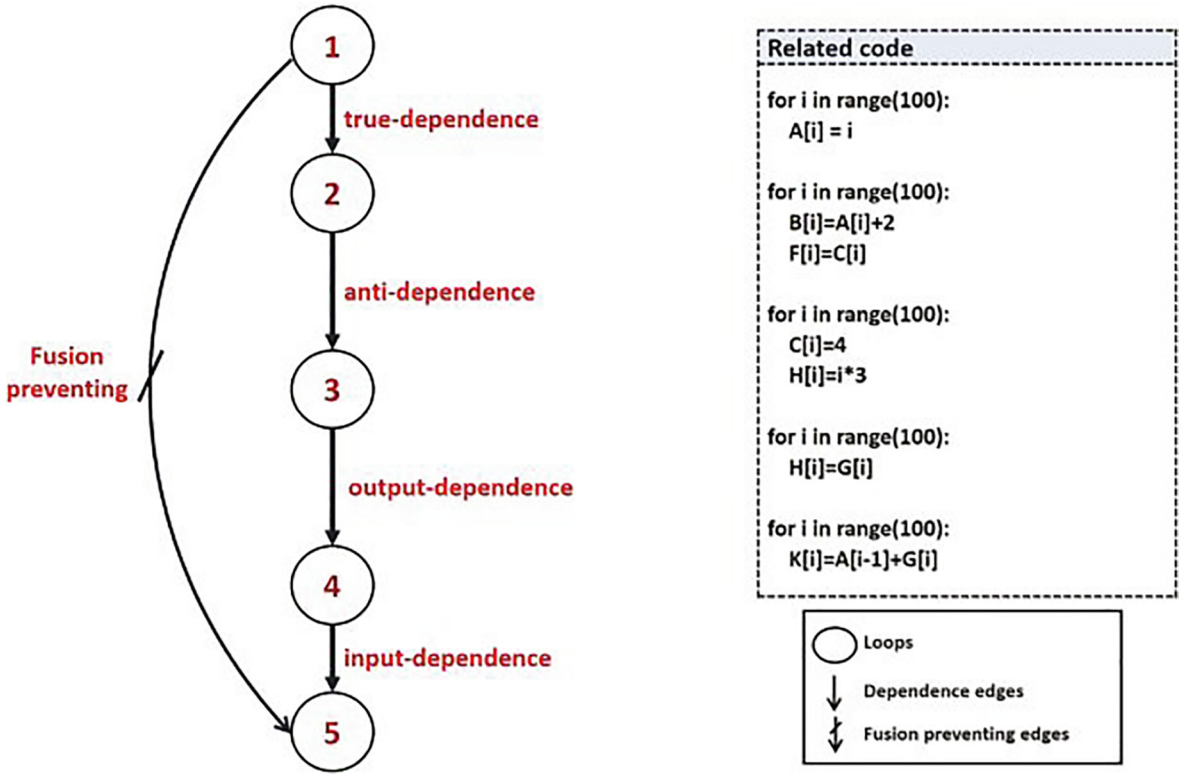


Fig. 2. Individual types of data dependencies.

tionary algorithm is first mentioned in [46], but it still needs developments to solve more intricate fusion problems. Beyond it, machine learning has not been exploited in the matter, despite its low execution time after training.

As an instance, [57,58] can be referred to as remonstrative papers that take a pivotal step to introduce deep neural networks into this field and demonstrate how to use them for acquiring thread coarsening factor and heterogeneous mapping. Howbeit, there are some obstacles in applying this strategy to the loop fusion problem. The nature of the fusion problem causes its output to be formed based on its input. On the other hand, training such networks requires a large amount of labeled training data, which is infeasible in this NP-hard problem. Over and above, it is not beneficial to tie an algorithm's performance to the quality of the labeled data whereas a method like reinforcement learning needs no labels. In reinforcement learning [59,60], the problem is introduced to the agent through the Markov decision process, and then it is deciphered by specifying the evaluation function. In fact, it is the agent that, by trial and error, tries to find the most acceptable policy for solving this problem by maximizing the reward prescribed from the environment based on the predetermined evaluation process [61,62]. In other words, the agent attempts to acquire a goal by taking into account the entanglement with the environment and the feedback receiving from it. After each action, the agent changes to a new state in the environment and receives a reward premised on the action performed and the intended goal [60,61]. The disadvantage of this approach is its demand to delineate the environment for it, in other words, the problem space. When factors such as register size and loop type are supplemented to the circuit, the problem space increases, accordingly cannot be defined in terms of the Markov decision process. More precisely, two methods can be used, finding a direct mapping from state space to action space or applying a function to measure the state's property for guiding the agent in its decisions [63]. For problems such as fusion, which has many parameters involved and therefore a large state space, the deep neural network is indispensable to model the problem by estimating the corresponding function. It implies that the proposed algorithm learns how to model the problem and the policy needed to solve it with deep reinforcement learning via deep neural networks, layers of which take the burden of modeling the problem [64]. Particularly, [63,65] are two influential papers that introduce deep reinforcement learning to solve the Traveling Salesman problem. Due to the similarity of the loop fusion and Traveling Salesman problems, this paper takes its idea from the indicated articles and, for the first time, proposes deep reinforcement learning for tackling the fusion problem benefiting from the framework presented in mentioned articles.

3.1. The proposed algorithm

According to the issues raised so far, loop fusion is a sequencing problem in which the best order of loops for merging must be determined, leading to an NP-hard problem. In this study, a generic framework is suggested for this challenge with the primary objective of ensuring the high compatibility of efficient machine learning algorithms. In this regard, a special edition of the adjacency matrix is created based on the information of the program and the related computer system. These matrixes are exploited as inputs of a deep neural network which is trained by reinforcement learning to attain a profitable order for fusing loops. The results manifest this approach's comparable performance with other existing methods. Fig. 3 illustrates its prevailing structure.

Concisely, a collection of basic information about the problem ahead, involving the type of loops, the data dependencies between them and the number of loop's arrays, as well as the number of arrays that the considered computer system's register can store, is prepared as a matrix which is given to the deep reinforcement learning network as an input. At this point, the agent evaluates

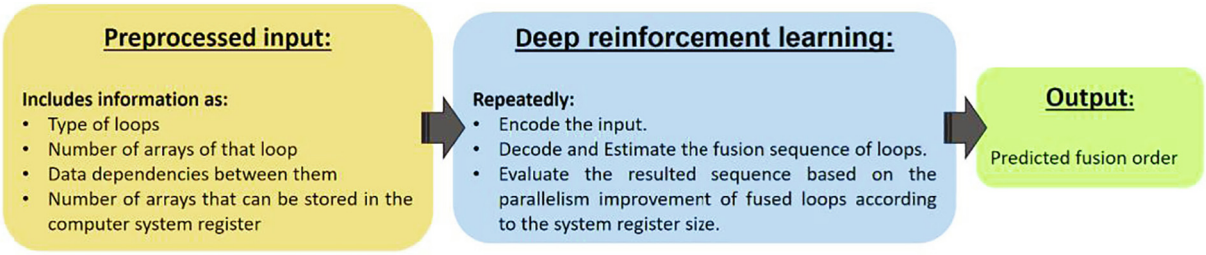


Fig. 3. The framework of the proposed model.

the quality of its results by repeating the fusion sequence estimation for that input and evaluating it based on factors such as parallelism advancement according to the system's register size. Later, it tries to procure better estimates for the successive inputs by using these experiences. All these steps are fully described in the next section.

4. Experimental results

As mentioned earlier, the DRLLF method is the first to introduce deep reinforcement learning for solving the complex loop fusion problem so that it can solve it in the shortest possible time. This section will describe in detail how this method works and the results obtained from it.

4.1. Experiments

In the proposed deep reinforcement learning context, the state is a half-done sequence of fused loops, and the action is choosing the next loop to fuse within not considered ones. During the learning process, the agent gains extensive experience in determining the appropriate policy for obtaining the optimal solution of the problem in question. In the continuation of this section, each component of this framework and its implementation is described in detail.

- **The neural architecture:**
  Considering the loop fusion problem, the proposed algorithm seeks to find a sequence of loops that, if they are merged as specified by the sequence, the outcome's execution cost is minimal; in other terms, the runtime of the resulting loops is optimal. These performance principles are ideal when the number of attained loops is augmented. By modeling the procedure outlined in [63,65], the intended method has learned the parameters θ of an appropriate policy for solving this problem and finding the proper fusion order by virtue of neural network and policy gradient. As the general encoder-decoder networks, in the first place, the required information is mapped to a specimen input matrix, owing to the use of which the output sequence is generated gradually.

Fig. 4 offers an overview of the intended process, each section of which is demonstrated schematically along with an example output related to it in a square to the right of that step. These components will be described in detail throughout the following sections.

- **Input preprocessing:**
  The input of a machine learning process stays ahead in its performance, especially in deep reinforcement learning, because they are supposed to provide all the essential information. Therefore, inputs should be able to reflect all the distinguishing features that are going to guide the agent in the learning procedure of an impeccable policy. Basically, these components are along the lines of the criteria settled out in Section 3. Hence, a matrix has been designed to accommodate all of these values, picturing in Fig. 5.

All the imperative data bound up in the problem is retained as a matrix holding floating-point numbers. The main diagonal of the matrix contains information about the type of each loop and the register size of the computer system on which the loops will be executed. More precisely, the integer part indicates the size of the register, and the fractional part defines the corresponding number to the loop's type. To specify the type number of nodes, they are labeled based on being sequential or parallel, their number of levels, and boundaries. The other matrix elements provide details about various dependencies between the loops. In each row, the integer part indicates the type of dependence, elaborated in Table 2, and the fractional part implies the number of register units required to execute that loop. According to Table 2, values that are not in the main diagonal of the matrix are quantified based on properties such as the topology of the graph, types of dependencies, and types of loops. By way of illustration, if it is topologically impossible to place two loops, as i and j, in the output sequence consecutively, the integer parcel of the pertinent data in row i and column j in the input matrix is set to 10.

Fig. 6 elucidate the formation of an input matrix. In the corresponding graph, the type of each node is illustrated with a distinct shape and the dependencies between them with an edge, also fusion preventing dependencies by an edge with a dash. Accordingly, 2.01 and 2.02 in the main diagonal of the matrix denote the parallel (0.01) and sequential (0.02) type of the examined loops, and the computer register size of 2 units. Moreover, 1.1 in the first row of the third column implies a data dependence between these two nodes of the same type, 1 and 3, beside the one register unit requirement for executing the loop number 1. Alternatively, number 10.1 in the fourth row and second column points out that it is topologically incorrect for node number 2 to come after node number 4 in the target sequence. In view of the fact that loop number 2 is loop 4′s parent in the dependence graph and thus has to appear before 4 in the loops' chain. Subsequently, a pattern like 1, 2, 3, 4, 5 could be a topological sequence for this graph. Similarly, 7.1 in the third column of the fourth row demonstrates that loop numbers 3 and 4 are not of the same type, and there is no data dependence between them. The pseudo-code related to random input matrix generation steps for the deep reinforcement learning network is demonstrated in Algorithm 1.

It should be noted that if the maximum number of loops is set to n during the training phase, in fusion problems with fewer loops the rest of the matrix cells are filled with values that reveal the absence of these nodes. In reference to the pseudo-code, during the input generation process, first, the pertinent parameters are valued, and then a random DAG is generated regarding them. On those grounds, the adjacency matrix of the corresponding graph is quantified with the desired values, in reliance on the type of loops involved and the dependencies between them.

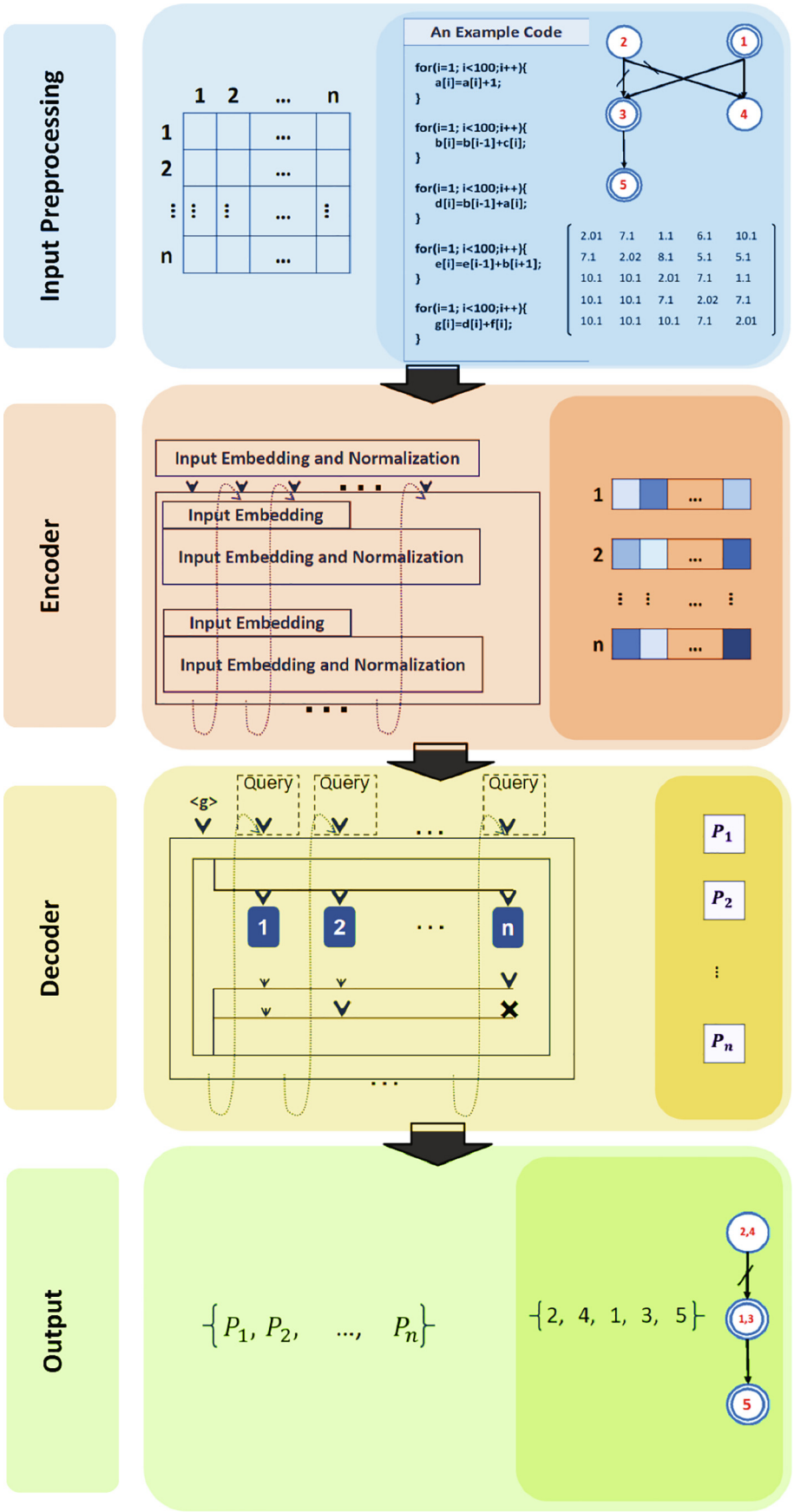


Fig. 4. The architecture of deep reinforcement learning scheme.

Algorithm 1: Random Input Generation

Input: V: maximum number of vertices; T: maximum number of loop types; R: system maximum register size;
Output: an instance of deep reinforcement learning algorithm's input

initialize:
G = a graph with a random number of vertices, limited to V;
D = a random number that specifies the number of the graph's edges;
Array T = randomly typed nodes (loops) based on the specified number of types (T);
Array R = Required register size of each node, determined randomly based on R;
RGSTR = system register size, determined randomly based on R;
for (the specified number of edges (D))
  edges are added to the graph (G) randomly, provided that the resulting graph is a DAG;
end for
A = adjacency matrix related to graph G (a matrix with 0 or 1 values, the main diagonal is 0);
While (There is an unchecked edge in the adjacency matrix, in other words, data values equal to 1)
  if (the two corresponding nodes are of the same type)
    e = randomly determine that this edge is fusion preventing or not;
    if (e is fusion preventing)
      the corresponding value in the adjacency matrix A = 5.required register size (R);
    else
      the corresponding value in the adjacency matrix A = 1.required register size (R);
    end if
  end if
  if (the two corresponding nodes are not of the same type)
    e = randomly determine that this edge is fusion preventing or not;
    if (e is fusion preventing)
      the corresponding value in the adjacency matrix A = 6.required register size (R);
    else
      the corresponding value in the adjacency matrix A = 8.required register size (R);
    end if
  end if
end while
for (each data value in main diagonal of the adjacency matrix A)
  the related value in the adjacency matrix A = RGSTR. node type (T);
end for
for (other adjacency matrix data values that are equal to 0)
  if (the two corresponding nodes are of the same type)
    if (the two corresponding nodes can be topologically consecutive)
      the corresponding value in the adjacency matrix A = 3.required register size (R);
    else
      the corresponding value in the adjacency matrix A = 10.required register size (R);

* (*continued*)

Algorithm 1: Random Input Generation

    end if
  end if
  if (the two corresponding nodes are not of the same type)
    if (the two corresponding nodes can be topologically consecutive)
      the corresponding value in the adjacency matrix A = 7.required register size (R);
    else
      the corresponding value in the adjacency matrix A = 10.required register size (R);
    end if
  end if
end for

- **Encoder:**
  As mentioned in Fig. 4, the encoder receives loop specifications as the input, or in this case a set of embedded and batch normalized actions [66]. It elicits an imitation form that reflects the characteristics of each loop as well the correlation between them. The actor and critic part of the applied method benefited from the neural attention mechanism, as in [63,67]. The encoder network took advantage of a two-layer structure for n times.

Its first sublayer is a multi-head attention that collects the resultant of applying linear transformation and then nonlinear ReLU on each loop specifications. Put another way, entries are sets of queries and key-value pairs, and outcomes are the new representation of each loop. In this regard, the sets are mapped to h different subspaces and concatenated through a weighted sum of them. The weights concerned in this process are obtained by applying an affinity function between the queries and keys. In fact, the attention mechanism is as mentioned in Eqn 1:

$$Attention(Q;K;V) = softmax\left(\frac{QK^T}{\sqrt{d}}\right)V \tag{1}$$

in which Q, K, and V are n-cell vectors. The second sublayer is a feed-forward, comprising two position-wise linear transformations with a ReLU activation separating them.

A normalization is applied to outputs of each sublayer, take place by means of summing the corresponding sublayer's output and the output of applying the function implemented in the sublayer to the same output. Indeed, encoded loops produce the action space of the decoder. As shown in the right box of the encoder patch in Fig. 4, the outturn of this step is actually a set of vectors, each of which properly specifies a loop. Each data value of these vectors is quantified, relying on loops' traits, the association between them, and the output of the previous step.

- **Decoder:**
  To calculate the probabilities of the sequences, a chain rule is used as the Eqn 2 formula [65].

$$p_\theta(\pi|s) = \prod_{t=1}^{n} p_\theta(\pi(t)|\pi(<t).s) \tag{2}$$

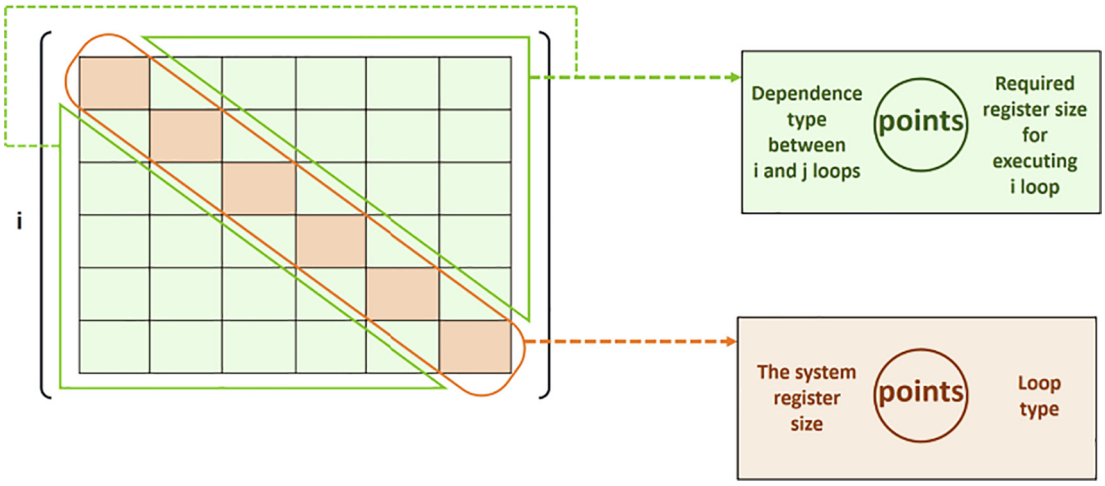


Fig. 5. Designation of input Matrix.

Table 2
The notion of the input matrix data values.

Dependence type	Assigned integer
Two dependent loops with the same type	1
Two not dependent loops with the same type	3
Two same type loops with a fusion preventing edge	5
Two dependent loops with different types	6
Two not dependent loops with different types	7
Two deferent types loop with a fusion preventing edge	8
Two loops that can not be topologically sequenced in fusion order	10

Qua discussed in the following formula [63], for anticipating the next loop in the sequence, at each time t, there are the last three added loops (actions) in the examination. In simple words, after every three steps, some information will be disregarded.

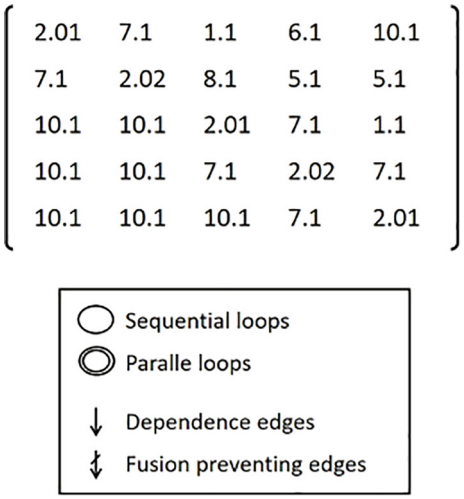
$$q_t = ReLU\left(W_1 a_{\pi(t-1)} + W_2 a_{\pi(t-2)} + W_3 a_{\pi(t-3)}\right) \in \mathbb{R}^d \tag{3}$$

As mentioned in Eqn 3, at each step of erecting the complete sequence, for identifying the distribution over the loops space, a query vector q_t is calculated based on the associated set of vectors. More precisely, this query vector forms a state representation in every single step. This allows the pointing mechanism to calculate the probability distribution on the remaining loops, receiving the encoded loops (actions) and a query (state representation), and decide on the next loop in the sequence. Thereby, as referred to in the below equations, each of these probabilities is computed sequentially via the Softmax module. Through this mechanism, it becomes feasible to address the fusion problem with a different number of loops. Two attention matrix and an attention vector are used to parameterize the pointing mechanism, regarding [65]:

$$\forall i \leqslant n.u_i^t = \begin{cases} v^T \tanh\left(w_{ref} a_i + w_q q_t\right) & \text{if } i \notin \pi(0)\ldots\pi(t-1) \\ -\infty & \text{otherwise.} \end{cases} \tag{4}$$

$$p_\theta(\pi(t)|\pi(<t).s) = softmax(C\tanh(u^t/T)) \tag{5}$$

To guarantee the exactness of the output sequences and absence of duplicated loops, a mask is used pursuant to [65] for settling the probability of choosing each loop, namely setting the likelihood of loops that have already appeared in the sequence to $-\infty$, as highlighted in equation Eqn 4. Besides, to manage the certainty of sampling, T works as a temperature hyper-parameter during training and inference, respectively valued with 1 and > 1 in Eqn 5. As illustrated in Fig. 4, the result of the decoder will be a predicted fusion order, an evaluation of which is going to be used for strengthening the proposed neural network's expertise and thus succeeding estimates.



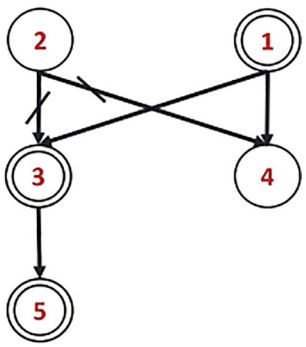


Fig. 6. A sample graph and its associated input matrix.

- **Training the model:**
  As discussed earlier, reinforcement learning allows the agent to gain the necessary experience for solving the fusion problem and find an optimal solution by examining the different sequences of loops on the basis of the related rewards. According to [65], the training phase is based on policy gradient, benefiting from reinforcement learning structure and a critic for reducing the variance of gradients. To attain this objective, a suitable reward function for the given graph s is defined in this paper that empowers the learning process by required information, as follows:

$$\begin{aligned} r(\pi|s) = {} & \textit{number of topological order violations in the considered sequence} \\ & + \textit{number of consecutive loops in the considered sequence that are not fusiable} \\ & + \textit{number of times that fusible loops' required register size passes the system's register size} \\ & + \textit{One tenth of the total weight of the edges between nodes in the sequence} \end{aligned} \tag{6}$$

In conformity with Eqn 6, a set of penalties are added to $r(\pi|s)$, and for this reason, the goal of this algorithm will be minimizing the penalties' amount. Hence, factors such as the number of times that the topological order in fusion sequence of loops is not observed, the number of times that two not fusible loops are placed consecutively, number of times that two fusible loops cannot be merged due to insufficient system register size, and the total weight of the edges between the loops in the fusion order which are determined by similarity degree in reliance on the type of loops and the type of edge across them, introduced in Table 2, affect the value of it. Deserving attention, if one of the two consecutive loops is not topologically ordered, they will not be merged, and the edge weight between them will be considered equal to 10 (conforming Table 2), even if the corresponding element in the matrix has a value apart from 10. Be careful that the impact of all these factors is one, except for total weights, which is one-tenth.

This formula, along with its relevant coefficients, has been acquired throughout various scrutiny, and its accuracy has been confirmed in practice. Fig. 7 represents this calculation process for an example predicted fusion order.

Considering the presented matrix and the output sequence, it is topologically incorrect that loop 4 appears before loop 1 in the referred order. Beyond that, loops 2 and 4 are not fusible due to the presence of a fusion preventing edge across them. Till now, two and one-tenth of the related edge's weight, which is ten because of wrong topological order, is considered as the penalty (3). Loops 4 and 1 cannot be merged due to their type variety, which is sequential and parallel, respectively, and also 4 is in the wrong topological order. Therefore, the corresponding penalty will be equal to one plus one-tenth of the edge's weight between them (2). There is a dependence edge between loops 1 and 3, having the same type. On the other hand, since the required register size for executing them is identical to the system's register size, which equals to 2, these two merged loops have only one-tenth weight of the edge between them as a penalty (0.1). Loop 5 can also be merged with loops 1 and 3, but since it requires an extra register unit and the system register is full, it cannot be fused with them, resulting in a penalty equal to one plus one-tenth of the related edge's weight (1.1). According to this content, the maximum and minimum possible value for a fusion problem with n loop would be:

$$(n-1) \times 0.1 \leq r(\pi|s) \leq ((n-1) \times 3) + 1 \tag{7}$$

As mentioned in Eqn 7, the minimum value occurs when all loops are of the same type and connected to each other by a dependence edge; in this way, all of them can be merged together. Taking one example, consider a Skewed Binary Tree as a dependence graph that its same typed nodes are ordered from root to leaf for fusion. Furthermore, the size of the system's register must be large enough to run all the fused loops together. In the present case, the expected value would be equal to one-tenth of the edges between the loops' weight sum, each of which is going to be 1. Conversely, this value would be maximum if none of the loops are topologically aligned; thus the across edge weight is equivalent to 10, and no two consecutive loops can be merged. In this way, three penalties for each pair of loops in the sequence and one penalty for the first loop's topology breach will be added to the relevant value. A Skewed Binary Tree that its nodes are fused from leaf to root could be reported as an instance. In reference to [63], the intention of reinforcement learning practice is accurately estimating the expected reward, which is defined as:

$$J(\theta|s) = \mathbb{E}_{\pi \sim p_\theta(.|s)}[r(\pi|s)] \tag{8}$$

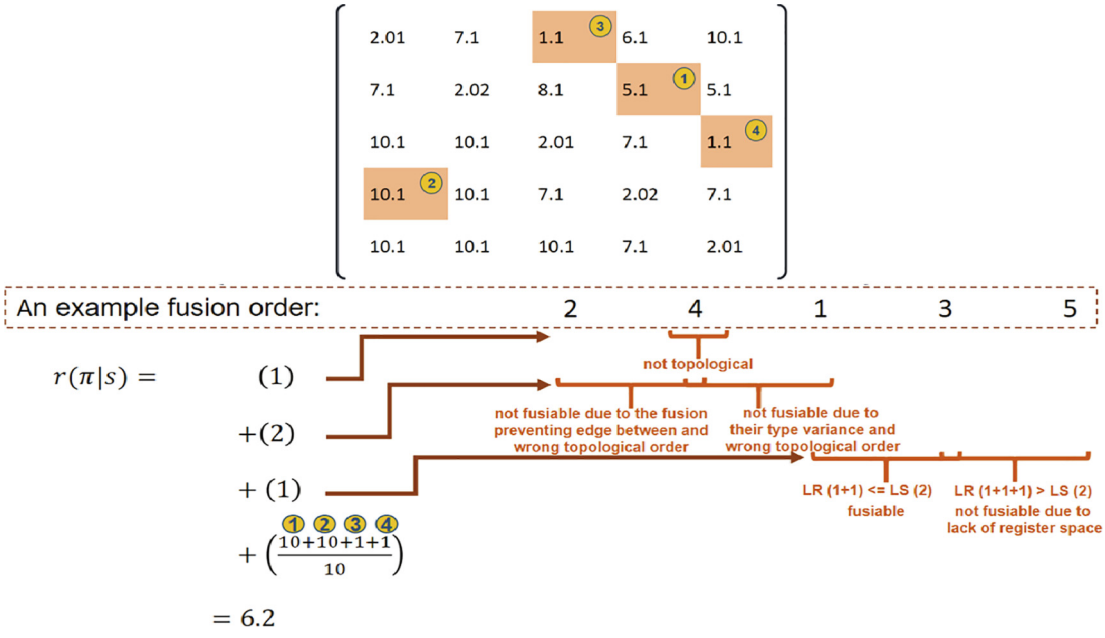


Fig. 7. The penalty calculation process of an example fusion order and its related matrix.

Thereby, as mentioned in Eqn 9, for distribution S:

$$J(\theta) = \mathbb{E}_{s\sim S}[J(\theta|s)] \tag{9}$$

To sort out the non-differentiability challenge of the hard-attention mechanism, [63] offered benefiting from the reinforcement learning rule of [68] and achieved an unbiased gradient as:

$$\nabla_\theta J(\theta|s) = \mathbb{E}_{\pi\sim p_\theta(.|s)}[(r(\pi|s) - b_\phi(s))\nabla_\theta \log(p_\theta(\pi|s))] \tag{10}$$

In Eqn 10, θ and $b_\phi(s)$ are the models' parameters and the baseline of critic, which reduces the variance of the gradients while maintaining them unbiased. The Monte-Carlo sampling approximates the gradient of Eqn 8 as Eqn 11 [63]:

$$\nabla_\theta J(\theta) \approx \frac{1}{B}\sum_{k=1}^{B}[(r(\pi_k|s_k) - b_\phi(s_k))\nabla_\theta \log(p_\theta(\pi_k|s_k)] \tag{11}$$

Table 3
Quantification of the proposed method's hyper-parameters.

Hyper-parameter's list	Initial values
Test set size	1000
Batch size	256
Number of loops in each input	8
Number of loops type in each input	4
Maximum number of system's register size in each input	10
Actor critic's each loop embedding dimension	128
Encoder inner layer neurons	512
Self-attentive encoder's number of stacks	3
Self-attentive encoder's number of parallel heads	16
Self-attentive encoder's number of hidden dimensions	128
Decoder or pointing mechanisms query space dimension	360
Decoder and critic attention product space	256
Critic's feed-forward layer	256 and 1 hidden unit
Number of steps	22,000
Critic initial baseline	7.0
Actor initial learning rate	0.001
Actor initial learning rate decay step	5000
Actor initial learning rate decay rate	0.96
Pointer initial temperature	1.0
Pointer tanh clipping	[−10.0, 10.0]
Feed-forward neural network input dimension	128
Feed-forward neural network output dimension	128
Feed-forward neural network inner-layer dimension	512

The whole process is accomplished via considering a random policy and gradually enhancing it in reliance on the received samples and evaluating them, assisted by reinforcement learning and Stochastic Gradient Descent (SGD).

The critic exploited in [63] is in common with the actor's encoder. The appointed glimpse vector is the weighted sum of loop's pointing distribution ($p_\phi(s)$) and action vector (a) multiplication as:

$$gl_s = \sum_{i=1}^{n} p_\phi(s)_i a_i \tag{12}$$

A network with two fully connected layers and ReLu activations receives the gl_s vector of Eqn 12 and tries to minimize the Mean Square Error during the prediction phase and the reward calculated by the actor.

Hyper-parameters of the proposed algorithm are quantified based on the values recorded in Table 3. The hyper-parameters related to the input properties are determined relying on the available examples for comparison. Namely, the maximum number of types in the papers being compared is 2; therefore, it is settled to 4 in this work to be able to compare it with them in addition to show its ability for more types. Besides, θ parameters are initialized using the Xavier-initializer [69], ensuring that the saturation of non-linear activation functions can be prevented and the scale of the gradients is kept the same in all layers. Adam [70] optimizer is used by Stochastic gradient descent in which $\beta_1 = 0.9$, $\beta_2 = 0.99$, and $\varepsilon = 10^{-9}$. The proposed model is implemented with Tensorflow and trained on an HP-Spectre laptop (Intel Core i7, 16 GB Memory) for approximately three weeks.

4.2. Results

For performance deliberation, this section compared the proposed method with investigations concerning the loop fusion problem. Toward that end, the proposed network is trained with a set of 1000 test cases, each member of which is a program with a maximum of 8 loops owing to the size of the available benchmarks for comparison. This means that several 8×8 matrices are randomly generated as input. After training the network, its capabilities are evaluated based on comparing its results with samples of related articles. It should be noted that the proposed method focuses on ameliorating loop fusion and therefore provides a comprehensive method for it, but recent articles in this field pay more attention to fusion's application or consider it along with other loop transfor-

Table 4
Comparing the proposed method with comparative works in terms of fusion sequence and the resulted loops.

[Ref.]	Original loops sequence	Referred paper's result	DRLLF's result
[31]	|1|2|3|4|5|6|7|8|	**|1-2|3-4|5-6|7-8|**	**|1-2|3-4|5-6|7-8|**
	Number of loops: 8	**Number of loops: 4**	**Number of loops: 4**
[32]	|1|2|3|4|5|6|7|8|	**|1-3|2-4-6|5-8|7|**	**|1-3|2-4-6|5-8|7|**
	Number of loops: 8	**Number of loops: 4**	**Number of loops: 4**
[33]	|1|2|3|4|5|6|7|8|	|1-2-8|3-4-5-6|7|	**|1-2-3-8|7-4-5-6|**
	Number of loops: 8	Number of loops: 3	**Number of loops: 2**
[8]	|1|2|3|4|5|6|	**|2|1-3|4-5|6|**	**|2|1-3|4-5|6|**
	Number of loops: 6	**Number of loops: 4**	**Number of loops: 4**
[35]	|1|2|3|4|5|	|1|2|3-4-5|	**|1-2|3-4-5|**
	Number of loops: 5	Number of loops: 3	**Number of loops: 2**
[36]	|1|2|3|4|5|6|7|	**|1-2-3–4-6|7|5|**	**|1-2|7|3-4-5-6|**
	Number of loops: 7	**Number of loops: 3**	**Number of loops: 3**
[41]	|1|2|3|4|5|6|	**|1-2-4|3-5-6|**	**|1-2-4|3-5-6|**
	Number of loops: 6	**Number of loops: 2**	**Number of loops: 2**
Ben6[41]	|1|2|3|4|5|6|7|	**|1-3|4-7|2-6|5|**	**|1-3|4-7|2-6|5|**
	Number of loops: 7	**Number of loops: 4**	**Number of loops: 4**
LL8[41]	|1|2|3|4|5|6|	**|1-2-3–4-5-6|**	|1-2-3|4-5-6|
	Number of loops: 6	**Number of loops: 1**	Number of loops: 2
LL18[41]	|1|2|3|4|5|6|	**|1-2|4-3-5-6|**	**|1-2|4-3-5-6|**
	Number of loops: 6	**Number of loops: 2**	**Number of loops: 2**

mations which will affect its result. Therefore, the number of articles whose results can be compared with the presented method is limited to Table 4 and Table 5, collecting the benchmarks and some examples from the cited papers with their features in terms of the number of loops and execution time. The loop dependence graphs of these samples are illustrated in Table A1, given in Appendix.

In the following tables, the results of the research are evaluated from two different perspectives. Table 4 assesses the output in terms of fusion sequence obtained from the compared methods and the resulting number of loops. In addition, Table 5 measures their execution time in sequential and parallel. It is worth mentioning that the main goal in this algorithm is improving the parallel runtime, but knowing the sequential execution time and comparing it with the parallel execution time, considering different number of processors, reveals the better parallelization performed. More precisely, the speedup improvement is a measure of the performed parallelization's quality, which must always be greater than one in order to make it admissible. The authors have to mention that [41] applied its method to Ben6, LL8, and LL18 benchmarks, which are available at [71,72]; therefore, the mentioned method's output for these samples has been compared with the results of the presented approach. The rest of the recorded instances are from the cited articles and their findings.

As recorded in Table 4, each loop is marked with its specified number in the column related to the fusion sequence, in particular, the original state contains separate loops, marked with a "|" between them, but after fusing them on the basis of the article understudy or the method presented in this article, some of them

Table 5
Comparing the proposed method with comparative works in terms of parallelism and execution time.

[Ref.]	Number of processors	Original code			Referred paper's result			DRLLF's result		
		sequential execution time	parallel execution time	speedup	sequential execution time	parallel execution time	speedup	sequential execution time	parallel execution time	speedup
[31]	2	11,038	9811	1.12	**10,220**	**8552**	**1.19**	**10,220**	**8552**	**1.19**
	3		9703	1.13		**7412**	**1.34**		**7412**	**1.34**
	4		9703	1.13		**7476**	**1.36**		**7476**	**1.36**
	5		9703	1.13		**7476**	**1.36**		**7476**	**1.36**
[32]	2	11,038	9715	1.13	**11,420**	**8869**	**1.28**	**11,420**	**8869**	**1.28**
	3		9605	1.14		**8359**	**1.36**		**8359**	**1.36**
	4		9601	1.14		**8300**	**1.37**		**8300**	**1.37**
	5		9601	1.14		**8287**	**1.37**		**8287**	**1.37**
[33]	2	29,342	16,651	1.76	28,317	**14,341**	**1.97**	**28,112**	14,371	1.95
	3		12,314	2.38		10,141	**2.79**		**10,121**	2.77
	4		10,421	2.81		8303	3.41		**8121**	**3.46**
	5		9421	3.11		7203	3.93		**6871**	**4.09**
[8]	2	10,632	6372	1.66	**10,222**	**5412**	**1.88**	**10,222**	**5412**	**1.88**
	3		5622	1.89		**4211**	**2.42**		**4211**	**2.42**
	4		5571	1.90		**3831**	**2.66**		**3831**	**2.66**
	5		5571	1.90		**3751**	**2.72**		**3751**	**2.72**
[35]	2	6827	4835	1.41	6417	4361	1.47	**6212**	**3451**	**1.80**
	3		4615	1.47		3511	1.82		**2591**	**2.39**
	4		4615	1.47		3111	2.06		**2201**	**2.82**
	5		4621	1.47		2891	2.21		**1981**	**3.13**
[36]	2	41,137	22,561	1.82	**40,317**	19,881	2.02	**40,317**	**19,871**	**2.02**
	3		16,781	2.45		14,041	2.87		**14,031**	**2.87**
	4		14,231	2.89		11,321	3.56		**11,311**	**3.56**
	5		12,671	3.24		9621	4.19		**9621**	**4.19**
[41]	2	10,454	5881	1.77	**9534**	**5081**	**1.87**	**9534**	**5081**	**1.87**
	3		4221	2.47		**3451**	**2.76**		**3451**	**2.76**
	4		3271	3.19		**2641**	**3.60**		**2641**	**3.60**
	5		2641	3.95		**2111**	**4.51**		**2111**	**4.51**
Ben6	[41]	2	10,037	6801		**14,465**	**8025**	**1.80**	**14,465**	**8025**
					1.47					
	1.80									
	3	6464		1.55	**5665**		**2.55**	**5665**		**2.55**
	4	6464		1.55	**4561**		**3.17**	**4561**		**3.17**
5	6464		**4001**	**3.61**	**4001**					
		1.55				**3.61**				
LL8	[41]	2	24,929	21,927		**23,799**	23,799	1	24,029	**21,017**
					1.13					
	1.14									
	3	20,807		1.19	23,799		1	**19,937**		**1.20**
	4	20,117		1.23	23,799		1	**19,407**		**1.23**
5	19,807		23,799	1	**19,107**					
		1.25				**1.25**				
LL18	[41]	2	10,812	6441		**10,204**	**5071**	**2.01**	**10,204**	**5071**
					1.67					
	2.01									
	3	4431		2.44	**3491**		**2.92**	**3491**		**2.92**
	4	3331		3.24	**2651**		**3.84**	**2651**		**3.84**
5	2771		**2161**	**4.72**	**2161**					
		3.90				**4.72**				

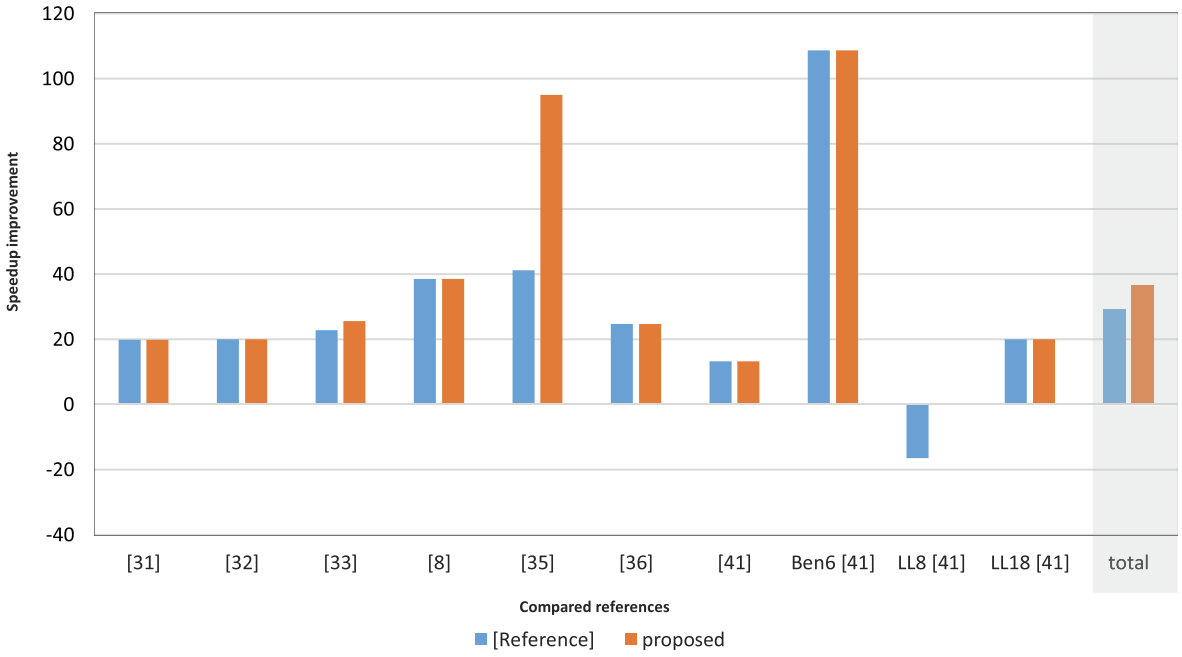
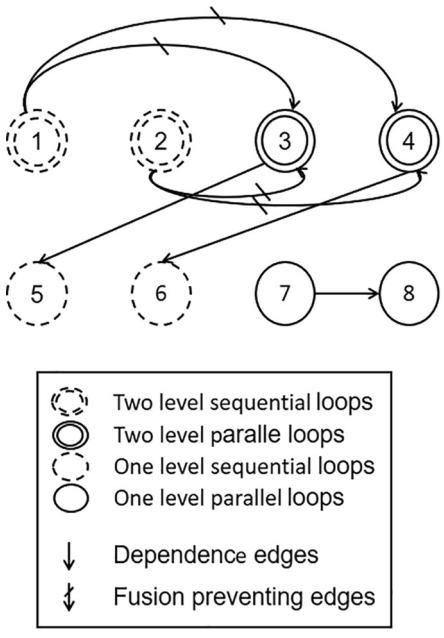


Fig. 8. The mean of speedup improvement rate.



Related code

```
for (i=0; i<100;i++){
   for (j=0; j<100; j++){
      a[i][j]= c+i-a[i-1][j+1];
   }
}
for (i=0; i<100; i++){
   for (j=0; j<100; j++){
      b[i][j]= 5*k+b[i-1][j];
   }
}
for (i=0; i<100; i++){
   for (j=0; j<100; j++){
      f[i][j]= a[i][j]+b[i][j];
   }
}
for (i=0; i<100; i++){
   for (j=0; j<100; j++){
      h[i][j]= a[i][j]+b[i][j];
   }
}
for (i=0; i<100; i++){
   m[i]= f[i][i]+m[i-1];
}
for (i=0; i<100; i++){
   n[i]= h[i][i]+n[i-1];
}
for (i=0; i<100; i++){
   g[i]= 4^i;
}
for (i=0; i<100; i++){
   t[i]= g[i]*2;
}
```

Fig. 9. A sample from the training phase to evaluate the performance in programs with four types of loops.

are merged, which is marked with a "-" sign between them. The resulted number of loops is mentioned below them.

This table compares the number of resulted loops based on the fusion sequences obtained. As demonstrated, by finding a better fusion order for the loops, the proposed method was able to obtain fewer loops in comparison to [33] and [35]. Only in LL8, the number of loops is more than the compared method, which is due to the fact that the loop's type is not considered in the mentioned article. In the proposed method, loops are fused according to their type, because merging two loops with different types will increase the execution time in the resulting code, which is not desirable. Considering LL8, fusing loops with different types may have reduced the final number of loops, but the execution time of DRLLF is better than [41], as stated in Table 5. In the other cases, the same results are obtained that due to the optimality of these answers, the algorithm's performance can be confirmed.

For a better inference, Table 5 compared the original program with the fusion result of the corresponding method and proposed fusion strategy in terms of their parallel or sequential execution time, given a system with 2 to 5 processors. Besides, the average speedup progression rate of the values listed in Table 5 is summarized in Fig. 8. The provided timing measurements are deliberated via a Multi-Pascal compiler (MPWinV.2.) which simulates the performance of the submitted code on a real multiprocessor system to produce its sequential and parallel execution time in microseconds [73]. Note that the calculated speedup is the ratio of serial execution time to parallel execution time as the Eqn 13 formula:

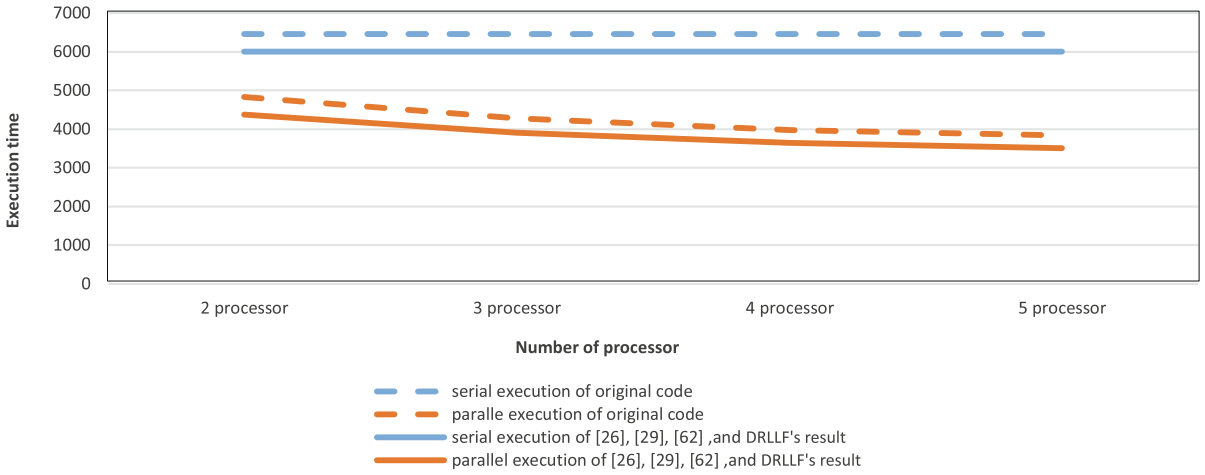
$$speedup = \frac{sequential\ execution\ time}{parallel\ execution\ time} \tag{13}$$

As stated in Table 5, the proposed method touched the same performance track of [8,31,32,41], Ben6 and LL18.Looking at the details, it could achieve a slight improvement in [36]. Though, the presented algorithm attains remarkable supremacy in [35,33], and LL8, marked by bolding the relevant row.

Demonstrated performance is the consequence of paying attention to concepts such as data dependence and fusion preventing edge that reduces runtime by raising data reuse, as well as considering diverse types for loops and not allowing to merge non-heterogeneous nodes. As an example, [41] has made the results of LL8 worse in terms of execution time because of assuming the existing nodes' type to be the same, in other words, not considering types for loops. This causes the diminution in the ability of parallel execution, and as specified in the relevant row of Table 5, the speedup is 1, implying that the run time of the program is equal in parallel and sequential mode. It is worth mentioning that given [33], the execution time results related to the two processor system of the proposed method slightly drop behind this article, but considering more processors makes it able to overtake the referred study with distance. This could owe to its lower accuracy in calculating data reuse because the relevant article examined the arrays inside the loops in this calculation. Account the fact that in 3 processors, the parallel and serial runtime of the proposed algorithm is better than the one compared, and the lower speedup is only because of its calculation method.

Fig. 8 affirms that in all test cases, the proposed algorithm was able to achieve the same or better performance in terms of average speedup rate, comparing with the methods under study, which is approximately averaged in 7.36 percent better results. This feature is calculated by dividing the speedup difference between the original code and the examined method to the original code speedup and finally obtaining the average of these values relating to the different number of processors.

To expose the capabilities of this algorithm in fusion problems with more than two number of loop types, a sample as one in Fig. 9 has been prepared, which is a random instance generated in the method training phase. In this manner, Fig. 10 reveals the progress rate obtained in the designed example, comparing the proposed algorithm with [32,36], and [46]. These articles have been selected due to the resemblance of the fusion problem sort under their study and taking into account more than two types



number of processors	2	3	4	5	fusion sequence
Original code speedup	1.33	1.51	1.62	1.68	|1|2|3|4|5|6|7|8|
[32] result's speedup	1.37	1.53	1.64	1.70	|1-2|3-4|5-6|7-8|
[36] result's speedup	1.37	1.53	1.64	1.70	|1-2|3-4|5-6|7-8|
[46] result's speedup	1.37	1.53	1.64	1.70	|1-2|3-4|5-6|7-8|
DRLLF result's speedup	1.37	1.53	1.64	1.70	|1-2|3-4|5-6|7-8|

Fig. 10. Runtime evaluation in terms of sequential and parallel execution, as well as speedup.

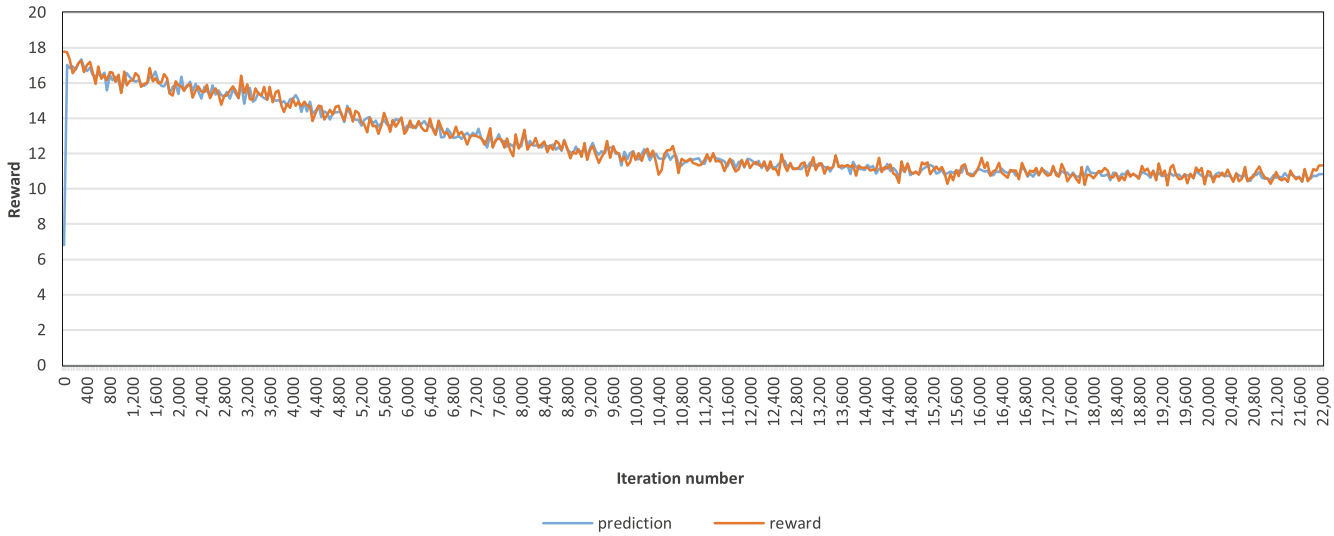


Fig. 11. Convergence diagram.

of loops. However, the slight absence of some features created the compulsion to design a new example for this purpose.

In more detail, the program under consideration includes four types of loops, mentioned in the Fig. 9 with their relative code.

In addition, it contains distinct kinds of dependencies, including fusion preventing edges. For this comparison, all the mentioned three methods have been implemented in Java and their outputs have been assessed.

The fusion order obtained for this sample in all cases is 1–2-3–4-5–6-7–8, as a result of which the following loops are merged, 1 with 2, 3 with 4, 5 with 6, and 7 with 8. By evaluating the graph and the result acquired, it can be concluded that this output is the optimal one for the examined problem. That is why, according to the assessment done via Multi-Pascal, the results have proven its superiority in terms of execution time.

4.2.1. Convergence and Stability

One of the critical factors in measuring the performance of distinct algorithms is the convergence assessment of results towards the optimal solution during the learning process. Fig. 11 depicts the convergence diagram of the proposed method, bringing to light that it has this feature. In this figure, the prediction indicates the average of the estimates made about the reward of each sample, and the reward implies the average actual reward of each of them. The trend of changes in these two factors throughout the learning process, and their values closeness to each other can be the consequence of the algorithm's convergence. Apart from it, considering that in repeating all the performed tests the algorithm has always been able to obtain the recorded results, the stability of this method can also be concluded. All these promising results support the reliability of the proposed method.

4.3. Discussion

Due to the fact that the loop fusion problem is a complex problem, the available methods have reduced its complexity by applying some assumptions to the problem, and thereby, they have succeeded in solving it in a short time. Actually, they have failed to address the main fusion problem. Some other papers have resorted to methods such as evolutionary algorithm, but their algorithms need a lot of time to find a suitable solution. In this paper, a new approach was presented for solving the fusion problem that, with the help of deep reinforcement learning, was able to solve this complex problem in the shortest possible time without applying any assumptions limiting the problem space. In addition to the quick response time of this method, the obtained results are also superior and confirm its performance. In most samples, the proposed method has been able to get the same results as the compared methods, and due to the optimality of these answers, the algorithm's performance can be confirmed. In other cases, the proposed method was able to achieve better performance than the compared methods, approximately averaged in 7.36 percent better results. The considerable improvement observed in the results, besides the low run time, proves the comprehensiveness and superiority of this approach.

5. Conclusion

Relying on the content raised so far, introducing deep reinforcement learning for the first time to this area brought a set of progression to the proposed algorithm. The capability of solving fusion problem with its complexities such as the number of loops' types, the variety of dependencies between them, and the characteristics of the system on which the loops are to be run as the register size, and beyond all this, finding the solution in the shortest time, are very few influential strengths that this method has been able to achieve. In contrast, this method has some shortcomings, namely not considering the arrays inside the loops to improve the evaluation of the fusion priority between the loops or not taking into account other loop transformations along with fusion to improve the results. Despite all the strengths and weaknesses of the proposed method, it could be a cornerstone in this area, owing to its abilities that are not limited to the mentioned ones, regarding its high development aptitudes that can be researched. For instance, this framework has a solid base for further progression in the following areas:

- More accurate calculation of data reuse by considering arrays of loops and their size.
- Taking into account other loop transformations, such as loop shifting and loop splitting, can increase the rate of improvement by removing some barriers to fusing loops.
- Including other transformations raises new issues in this problem, in particular, the order of applying these transformations, which can either be considered as a fixed order or the optimal order can be obtained. This question is referred to as a valuable research field.

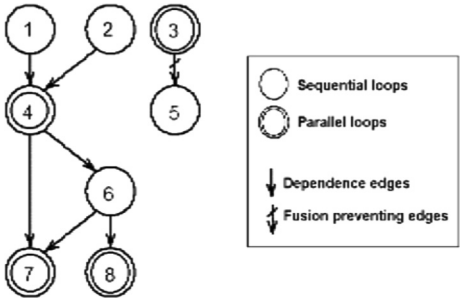
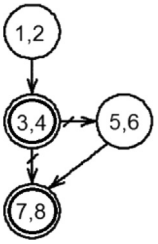
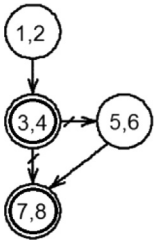
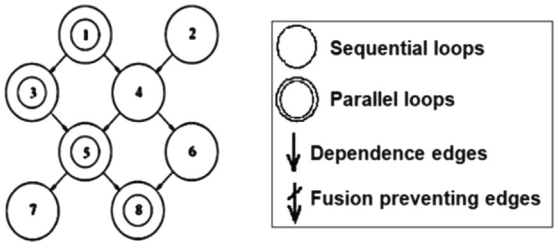
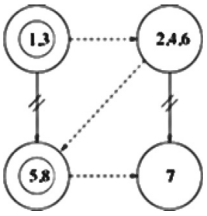
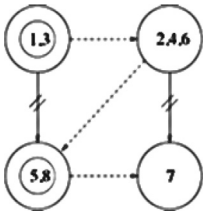
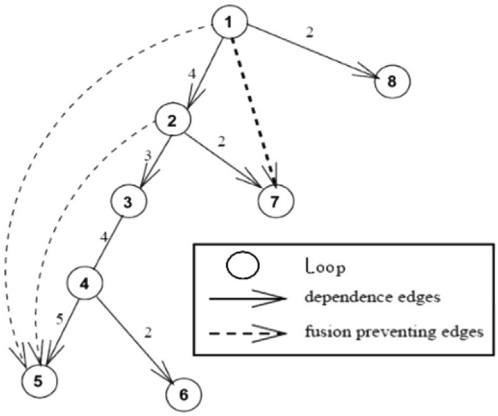
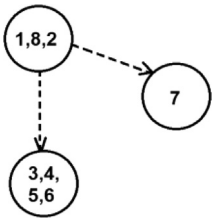
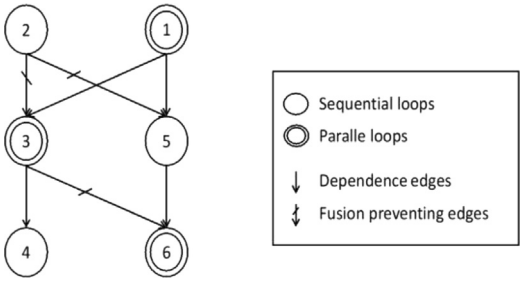
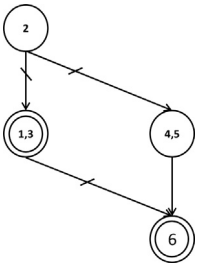
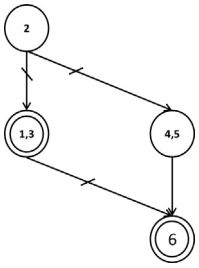
- The outcomes of deep reinforcement learning methods depend on their evaluation function. According to the discussed results, the presented evaluation function can be improved. For instance, the execution time of a program can be the best and most comprehensive criterion for this work because it includes many other criteria implicitly. On the other hand, the primary goal of all the existing methods is to improve the execution time. Although it is necessary to point out that estimating this value is also a field of research.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

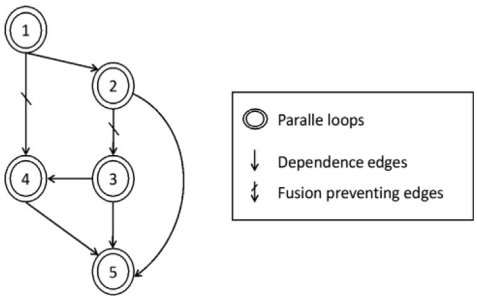
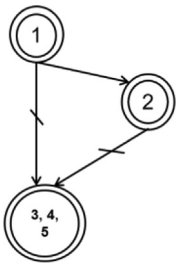
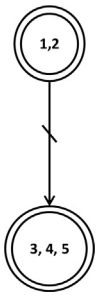
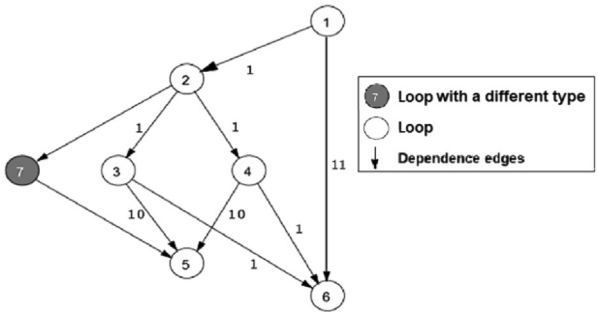
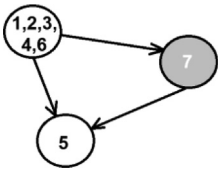
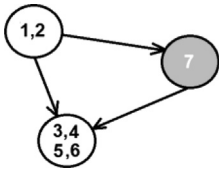
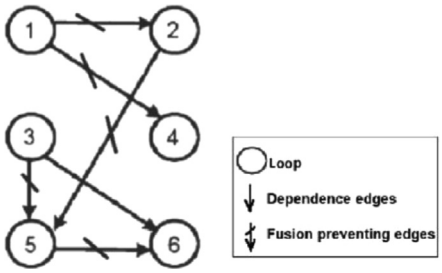
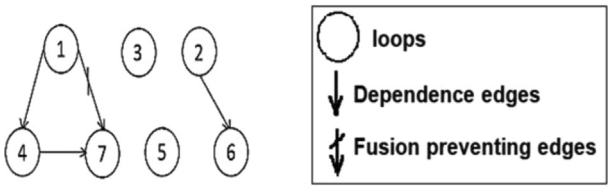
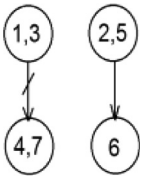
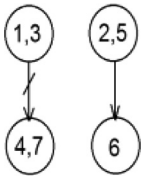
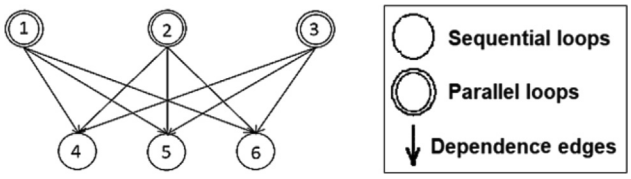
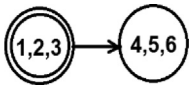
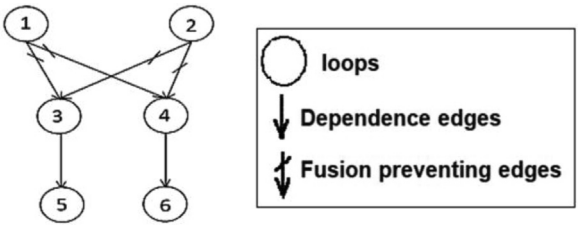
Appendix

Table A1
Loop dependence graph of the represented results in Table 4.

[Ref.]	Original dependence graph	Referred paper's result	DRLLF's result
[31]			
[32]			
[33]			
[8]			

(continued on next page)

Table A1 (*continued*)

[Ref.]	Original dependence graph	Referred paper's result	DRLLF's result
[35]			
[36]			
[41]			
Ben6 [41]			
LL8 [41]			
LL18 [41]			

References

[1] E. H. M. Sha, N. L. Passos. Efficient polynomial-time nested loop fusion with full parallelism. (1999).

[2] S. Blom, S. Darabi, M. Huisman, Verification of loop parallelisations, in: International conference on fundamental approaches to software engineering, Springer, Berlin, Heidelberg, 2015, pp. 202–217.

[3] Q. Yi, K. Kennedy, Improving memory hierarchy performance through combined loop interchange and multi-level fusion, Int. J. High Perform. Comput. Appl. 18 (2) (2004) 237–253.

[4] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S.V. Adve, C.T. Chou, DeNovo: Rethinking the memory hierarchy for disciplined parallelism, in: 2011 International Conference on Parallel Architectures and Compilation Techniques, IEEE, 2011, pp. 155–166.

[5] Z. Jie, Z. Rongcai, Y. Yuan, A nested loop fusion algorithm based on cost analysis, in: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, IEEE, 2012, pp. 1096–1101.

[6] M. Liu, E.H.M. Sha, Q. Zhuge, Y. He, M. Qiu, Loop distribution and fusion with timing and code size optimization, J. Signal Process. Syst. 62 (3) (2011) 325–340.

[7] J. Cong, P. Zhang, Y. Zou, Combined loop transformation and hierarchy allocation for data reuse optimization, in: 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), IEEE, 2011, pp. 185–192.
[8] A. Darte, On the complexity of loop fusion, Parallel Comput. 26 (9) (2000) 1175–1193.
[9] M. Liu, Q. Zhuge, Z. Shao, C. Xue, M. Qiu, E.M. Sha, Maximum loop distribution and fusion for two-level loops considering code size, 8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05), IEEE, 2005.
[10] M. Ziraksima, S. Lotfi, H. Izadkhah, Loop fusion methods: a critical review, Int. J. Adv. Res. Comput. Commun. Eng. 6 (7) (2017) 1–8.
[11] J.C. Pichel, D.B. Heras, J.C. Cabaleiro, F.F. Rivera, Performance optimization of irregular codes based on the combination of reordering and blocking techniques, Parallel Comput. 31 (8–9) (2005) 858–876.
[12] S. Carr, C. Ding, P. Sweany, Improving software pipelining with unroll-and-jam, in: Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences, IEEE, 1996, pp. 183–192.
[13] Q. Yi, J. Guo, Extensive parameterization and tuning of architecture-sensitive optimizations, Procedia Comput. Sci. 4 (2011) 2156–2165.
[14] S. Lotfi, S. Parsa, Parallel loop generation and scheduling, J. Supercomput. 50 (3) (2009) 289–306.
[15] S. Parsa, S. Lotfi, A new genetic algorithm for loop tiling, J. Supercomput. 37 (3) (2006) 249–269.
[16] E. Hammami, Y. Slama, An overview on loop tiling techniques for code generation, in: 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), IEEE, 2017, pp. 280–287.
[17] N. Manjikian, T.S. Abdelrahman, Fusion of loops for parallelism and locality No. 2, in: IEEE transactions on parallel and distributed systems, 1997, pp. 193–209.
[18] I. Ştirb, H. Ciocârlie, Improving performance and energy consumption with loop fusion optimization and parallelization, in: 2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI), IEEE, 2016, pp. 000099–000104.
[19] A. Acharya, U. Bondhugula, A. Cohen, Effective Loop Fusion in Polyhedral Compilation Using Fusion Conflict Graphs, ACM Trans. Archit. Code Optimization (TACO) 17 (4) (2020) 1–26.
[20] M. Liu, Q. Zhuge, Z. Shao, E.H.M. Sha, September). General loop fusion technique for nested loops considering timing and code size, in: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems, 2004, pp. 190–201.
[21] P. Boulet, A. Darte, G.A. Silber, F. Vivien, Loop parallelization algorithms: from parallelism extraction to code generation, Parallel Comput. 24 (3–4) (1998) 421–444.
[22] G. Roth, K. Kennedy, Loop fusion in high performance Fortran, in: Proceedings of the 12th international conference on Supercomputing, 1998, pp. 125–132.
[23] B. Blainey, C. Barton, J.N. Amaral, Removing impediments to loop fusion through code transformations, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, Berlin, Heidelberg, 2002, pp. 309–328.
[24] P.S. Rawat, A. Sukumaran-Rajam, A. Rountev, F. Rastello, L.N. Pouchet, P. Sadayappan, Associative instruction reordering to alleviate register pressure, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 590–602.
[25] G. Gao, R. Olsen, V. Sarkar, R. Thekkath, Collective loop fusion for array contraction, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, Berlin, Heidelberg, 1992, pp. 281–295.
[26] A. Jangda, U. Bondhugula, An effective fusion and tile size model for optimizing image processing pipelines, ACM SIGPLAN Notices 53 (1) (2018) 261–275.
[27] M.R. Kristensen, S.A. Lund, T. Blum, J. Avery, Fusion of parallel array operations, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, 2016, pp. 71–85.
[28] A. Qasem, K. Kennedy, Profitable loop fusion and tiling using model-driven empirical search, in: Proceedings of the 20th annual international conference on Supercomputing, 2006, pp. 249–258.
[29] J. Xue, Q. Huang, M. Guo, Enabling loop fusion and tiling for cache performance by fixing fusion-preventing data dependences, in: 2005 International Conference on Parallel Processing (ICPP'05), IEEE, 2005, pp. 107–115.
[30] J. Warren, A hierarchical basis for reordering transformations, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM, 1984, pp. 272–282.
[31] K. Kennedy, K.S. McKinley, Maximizing loop parallelism and improving data locality via loop fusion and distribution, in: International Workshop on Languages and Compilers for Parallel Computing, Springer, Berlin, Heidelberg, 1993, pp. 301–320.
[32] K. Kennedy, K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. (1994).
[33] S.K. Singhai, K.S. McKinley, A parametrized loop fusion algorithm for improving parallelism and cache locality, Comput. J. 40 (6) (1997) 340–355.
[34] N. Megiddo, V. Sarkar, Optimal weighted loop fusion for parallel programs, in: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures, ACM, 1997, pp. 282–291.
[35] A. Fraboulet, K. Kodary, A. Mignotte, Loop fusion for memory space optimization, in: Proceedings of the 14th international Symposium on Systems Synthesis, 2001, pp. 95–100.
[36] K. Kennedy, Fast greedy weighted fusion, Int. J. Parallel Prog. 29 (5) (2001) 463–491.
[37] S. Verdoolaege, M. Bruynooghe, G. Janssens, F. Catthoor, Multidimensional incremental loop fusion for data locality, in: Application-Specific Systems, Architectures, and Processors, 2003. Proceedings. IEEE International Conference on, IEEE, 2003, pp. 17–27.
[38] C. Ding, K. Kennedy, Improving effective bandwidth through compiler enhancement of global cache reuse, J. Parallel Distrib. Comput. 64 (1) (2004) 108–134.
[39] P. Marchal, J.I. Gómez, F. Catthoor, Optimizing the memory bandwidth with loop fusion, in: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2004, pp. 188–193.
[40] M. Qiu, E.H.M. Sha, M. Liu, M. Lin, S. Hua, L.T. Yang, Energy minimization with loop fusion and multi-functional-unit scheduling for multidimensional DSP, J. Parallel Distrib. Comput. 68 (4) (2008) 443–455.
[41] W. Tian, C.J. Xue, M. Li, E. Chen, Loop fusion and reordering for register file optimization on stream processors, J. Syst. Softw. 85 (7) (2012) 1673–1681.
[42] S. Mehta, P.H. Lin, P.C. Yew, Revisiting loop fusion in the polyhedral framework No. 8, in: ACM SIGPLAN Notices, ACM, 2014, pp. 233–246.
[43] S.J. Bigdello, M.J. Bigdello, H. Mohammadzadeh, Improving efficency of mathematical functions in image processing by loop fusion, in: 2018 5th International Conference on Electrical and Electronic Engineering (ICEEE), IEEE, 2018, pp. 334–339.
[44] A. Imanishi Imanishi, K. Suenaga, A. Igarashi, A guess-and-assume approach to loop fusion for program verification, in: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, 2017, pp. 2–14.
[45] B. Qiao, O. Reiche, F. Hannig, J. Teich, From loop fusion to kernel fusion: a domain-specific approach to locality optimization, in: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), IEEE, 2019, pp. 242–253.
[46] M. Ziraksima, S. Lotfi, H. Izadkhah, Using an evolutionary approach based on shortest common supersequence problem for loop fusion, Soft. Comput. 24 (10) (2020) 7231–7252.
[47] A.E. Eiben, J.E. Smith, Introduction to Evolutionary Computing, Springer, Berlin, 2003, p. 18.
[48] K.J. Räihä, E. Ukkonen, The shortest common supersequence problem over binary alphabet is NP-complete, Theoret. Comput. Sci. 16 (2) (1981) 187–198.
[49] P.J. Radoszewski, T. Kociumaka, S.P. Pissis, W. Rytter, J. Straszynski, T. Walen, W. Zuba, Weighted Shortest Common Supersequence Problem Revisited, in: String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings, Springer Nature, 2019, p. 221.
[50] C. Gagné, W.L. Price, M. Gravel, Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence-dependent setup times, J. Operat. Res. Society 53 (8) (2002) 895–906.
[51] W. Deng, J. Xu, H. Zhao, An improved ant colony optimization algorithm based on hybrid strategies for scheduling problem, IEEE Access 7 (2019) 20281–20292.
[52] D.L. Applegate, R.E. Bixby, V. Chvátal, W.J. Cook, The Traveling Salesman Problem, Princeton University Press, 2011.
[53] E. Osaba, J. Del Ser, A. Sadollah, M.N. Bilbao, D. Camacho, A discrete water cycle algorithm for solving the symmetric and asymmetric traveling salesman problem, Appl. Soft Comput. 71 (2018) 277–290.
[54] K. Kennedy, J.R. Allen, Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, Morgan Kaufmann Publishers Inc., 2001.
[55] W.A.K. Abu-Sufah, Improving the Performance of Virtual Memory Computers, University of Illinois at Urbana-Champaign, 1979.
[56] H. Zima, B. Chapman, Automatic Restructuring for Parallel and Vector Computers, CRC Press, 2020, pp. 135–168.
[57] C. Cummins, P. Petoumenos, Z. Wang, H. Leather, End-to-end deep learning of optimization heuristics, in: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2017, pp. 219–232.
[58] Z. Wang, M. O'Boyle, Machine learning in compiler optimization, Proc. IEEE 106 (11) (2018) 1879–1901.
[59] S.S. Keerthi, B. Ravindran, A tutorial survey of reinforcement learning, Sadhana 19 (6) (1994) 851–889.
[60] K. Arulkumaran, M. P. Deisenroth, M. Brundage, A. A. Bharath. A brief survey of deep reinforcement learning. arXiv preprint arXiv:1708.05866. 2017.
[61] L.P. Kaelbling, M.L. Littman, A.W. Moore, Reinforcement learning: A survey, J. Artificial Intelligence Res. 4 (1996) 237–285.
[62] B. Recht, A tour of reinforcement learning: the view from continuous control, Annu. Rev. Control, Robotics, Autonomous Syst. 2 (2019) 253–279.
[63] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, L.M. Rousseau, Learning heuristics for the tsp by policy gradient, in: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer, Cham, 2018, pp. 170–181.
[64] Y. Li. Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274. (2017).
[65] I. Bello, H. Pham, Q. V. Le, M. Norouzi, S. Bengio. Neural combinatorial optimization with reinforcement learning. arXiv preprint arXiv:1611.09940. (2016).
[66] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: International conference on machine learning, PMLR, 2015, pp. 448–456.
[67] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, et al., Attention is all you need, in: Advances in neural information processing systems, 2017, pp. 5998–6008.

[68] R.J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Mach. Learning 8 (3-4) (1992) 229–256.
[69] X. Glorot, Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. in: Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256). JMLR Workshop and Conference Proceedings, 2010.
[70] D. P. Kingma, J. Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. (2014).
[71] F. McMahaon. The benchmark "Livermore". (2016). Available online at http://www.netlib.org/benchmark/. Accessed on Dec 2016.
[72] L. Wang, W. Tembe, S. Pande, A framework for loop distribution on limited on-Chip memory processors, in: International Conference on Compiler Construction, Springer, Berlin, Heidelberg, 2000, pp. 141–156.
[73] B.P. Lester, The Art of Parallel Programming, Prentice-Hall Inc., 1993.

Mahsa Ziraksima is majored from the Department of Computer Science at University of Tabriz, Iran with B.Sc. and M.Sc. in Computer Science. She is currently a Ph.D. candidate in Computer Science at University of Tabriz. Her research interests are optimizing compilers, evolutionary computation and machine learning.

Shahriar Lotfi received the B.Sc. in Software Engineering from University of Isfahan, Iran, the M.Sc. degree in Software Engineering from University of Isfahan, Iran, and Ph.D. degree in Software Engineering from Iran University of Science and Technology, Iran. He is currently an Associate professor of computer Science at the University of Tabriz. His research interests include compilers, super-compilers, parallel algorithms, evolutionary algorithms, social networks and algorithms.

Jafar Razmara is an Associate professor at Department of Computer Science, University of Tabriz. He received his B.Sc. and M.Sc. degrees in Computer Engineering from Isfahan University of Technology and Tarbiat Modares University, respectively. In 2012, he obtained his Ph.D. degree in Computer Science from Universiti Teknologi Malaysia. He was previously with the Faculty of Computing, Universiti Teknologi Malaysia as a senior lecturer. His main research interests are data mining, soft computing, and machine learning applications especially in bioinformatics and computational biology.