Reinforcement Learning assisted Loop Distribution for Locality and Vectorization

Conference Paper · November 2022

DOI: 10.1109/LLVM-HPC56686.2022.00006

7 authors, including:

Shalini Jain
Indian Institute of Technology Hyderabad
7 PUBLICATIONS **82** CITATIONS
SEE PROFILE

s. Venkatakeerthy
Indian Institute of Technology Hyderabad
10 PUBLICATIONS **90** CITATIONS
SEE PROFILE

Rohit Aggarwal
Indian Institute of Technology Hyderabad
5 PUBLICATIONS **74** CITATIONS
SEE PROFILE

Dibyendu Das
Intel
37 PUBLICATIONS **128** CITATIONS
SEE PROFILE

Reinforcement Learning assisted Loop Distribution for Locality and Vectorization

Shalini Jain*, S. VenkataKeerthy†, Rohit Aggarwal‡, Tharun Kumar Dangeti§, Dibyendu Das¶ and Ramakrishna Upadrasta‖
*†‡§‖Department of Computer Science and Engineering, IIT Hyderabad, ¶Intel India
Email: {*cs15resch11010, †cs17m20p100001}@iith.ac.in,
{‡rohitaggarwal0007, §dangeti.tharunkumar}@gmail.com, ¶dibyendu.das@intel.com,
‖ramakrishna@cse.iith.ac.in

***Abstract*—Loop distribution (loop fission) is a well known compiler optimization that splits the loop into multiple loops. Loop distribution can be seen as an enabler of various other optimizations with different goals, like, the parallelizability of the loop, the vectorizability of the loop, its locality characteristics.**

In this work, we present a Reinforcement Learning (RL) based approach to efficiently perform loop-distribution with the twin goals of optimizing for both vectorization as well as locality. Broadly, we generate the SCC Dependence Graph (SDG) for each loop of the program. Our RL model learns to predict the distribution order of the loop by performing topological walk of the graph. The reward to our model is computed using the instruction cost and number of cache misses of the loop. As the RL models need data for training purposes, we also propose a novel strategy to extend the training set by generating new loops.

We show our results on x86 architecture on various benchmarks: from TSVC, LLVM-Test-Suite, PolyBench and PolyBench-NN. Our framework achieves an average improvement of 3.63% on TSVC, 4.61% on LLVM-Test-Suite Microbenchmarks, 1.78% on PolyBench and 1.95% on PolyBench-NN benchmark suites for execution time. The baseline is `O3` compiler option of LLVM. We also show the improvements of our method on other performance metrics like Instruction Per Cycle (IPC), Number of loops distributed & vectorized, and L1 cache performance.

***Index Terms*—Loop Distribution, Vectorization, Locality, Reinforcement Learning**

I. Introduction

Loops are a crucial part of programs that play an essential role in exploiting performance. Performing loop optimizations generally leads to improvements in the speedup of the input programs. Loop distribution, also called as loop fission in literature, is one such optimization which involves splitting a single loop into two or more loops, often to exploit the code for better cache locality and parallelization.

Modern architectures provide SIMD vector registers, helping in performing a special form of parallelization, called vectorization. Vectorization is the process of converting a scalar program, where an instruction operates on a data item, to a vector program where the same operation is performed over multiple data items. Vectorization requires instruction set support like AVX for X86, Altivec for PowerPC and Neon for ARM [HLF+18], [NZ08]. Modern compilers try to identify and automatically convert scalar operations to vector operations with the help of optimizing transformations like loop vectorization (that works on loops) or SLP (Superword Level Parallelism) vectorization that works on straight-line code [LOa], [NRZ06]. Often loops in programs are not vectorized due to inhibiting data dependences though parts of the loops may still be vectorizable. In such cases if loop distribution is carried out intelligently, it can expose better vectorization opportunities. Thus, loop distribution can be an enabling transformation for vectorization by isolating the impeding dependences without disturbing and sometimes even improving the inherent cache locality. However, data locality and vectorization are interrelated and there is always a possibility that vectorizing a program may have adverse effects on locality and vice versa. Hence, there is a trade-off between these two optimizations and it is complex to make a decision which one (or both/ or none) suits well for a given program.

TABLE I
Example for Loop Distribution: (a) Loop1 before loop distribution, (b) Loop1 is not vectorizable after loop distribution but it improves Locality due to less cache pollution, (c) Loop2 before Loop Distribution, and (d) Only first loop become vectorizable after loop distribution but it reduces locality

Before Distribution	After Distribution

`// (a) Loop1`
`int x[N], y[N], a[N];`
`for (int i=1; i<N; ++i)`
`{`
`  x[i+1] = x[i-1] + x[i+1];`
`  a[i+1] = (a[i-1] + a[i])`
`  /2.0;`
`}`
	`// (b) Loop1: Distributed`
	`int x[N], y[N], a[N];`
	`for (int i=1; i<N; ++i)`
	`  x[i+1] = x[i-1] + x[i+1];`
	
	`for (int i=1; i<N; ++i)`
	`  a[i+1] = (a[i-1] + a[i])`
	`  /2.0;`

`// (c) Loop2`
`int a[N], b[N], c[N], d[N];`
`for (int i=1; i<N; i++)`
`{`
`  a[i] += c[i] * d[i];`
`  b[i] = b[i-1] + a[i]`
`  + d[i];`
`}`
	`// (d) Loop2: Distributed`
	`int a[N], b[N], c[N], d[N];`
	`for (int i=1; i<N; i++)`
	`  a[i] += c[i] * d[i];`
	
	`for (int i=1; i<N; i++)`
	`  b[i] = b[i-1] + a[i]`
	`  + d[i];`

We can observe this behaviour in Table I where distributing the first loop does not make the loop vectorizable due to the dependences within the instruction, while it improves data locality because of less cache pollution. On the other hand, distributing the second loop into two loops enables

Identify applicable funding agency here. If none, delete this.

vectorization for the the first one as the new loop does not have any memory dependence, while it reduces data locality due to more cache pollution.

As stated earlier, vectorization opportunities in a loop with n statements can be enhanced by the loop distribution transformation. However, it is a challenge to find the portion of a loop that when distributed makes the loop vectorizable, as this search space is very large. It has been shown that this transformation is a NP-Hard problem.

In this work we propose to use machine learning approaches to perform loop distribution that improves/exposes more vectorization opportunities in addition to having code with better locality characteristics. Suppose we have n number of instructions inside a loop then there could be at most $n!$ possibilities for loop distribution. It is hard to compute these many combinations of distribution patterns for a large value of n. So, it is highly infeasible to generate these many combinations for each loop present in the dataset and train with a supervised learning approach. Hence, We use a Reinforcement learning (RL) approach [SB18] to solve this problem. Because RL models learn by a process of receiving rewards on every action taken, it is able to train systems to respond to unforeseen environments, making it more relevant to be applied in the domain of compiler optimizations. RL is an increasingly popular technique for problems with large complex spaces and has seen several applications in compiler optimizations in recent times [HAAW$^+$20], [MYP$^+$19]. Though deploying machine learning approaches in compiler frameworks is often overlooked due to their nature of generating incorrect results, in this paper, we propose the framework that is always guaranteed to generate semantically correct code. We achieve this by mimicking/imitating the maximal loop distribution algorithm proposed by Allen Kennedy [DRV00].

There are various ways to mathematically represent a program as an input to an ML model. One of them is a feature based representation, which consists of representing program characteristics as features in the form of a feature vector. Feature based representations require human expertise. This problem can be avoided by having an embedding-based representation like Code2Vec [AZLY19], NCC [BNJH18], or IR2Vec [VAJ$^+$20]. In this work, we use IR2Vec representation of the loops. IR2Vec is an LLVM-IR based vector representation which encodes features/program characterstics, flow information and semantics of the code. We discuss this in detail in Sec. II-D.

In this work, firstly, we generate reduced dependence graph (RDG) for a given loop. We find the Strongly Connected Components (SCC) of the generated RDG and form a SCC Dependence graph (SDG). Then we do a topological walk over each SCC node of an SDG while making a decision whether to distribute or merge the current node with the previously processed node. This decision is made by the RL agents. Our problem is modeled as a Markov Decision Process (MDP) problem with rewards. We have developed an analytical throughput estimation model that takes into account locality and vectorization to compute these rewards. We use the LLVM infrastructure for modelling our methodology. We use IR2Vec to represent the instructions corresponding to the nodes of the SDG.

In summary, we make the following contributions:

- We model loop distribution as a Reinforcement Learning (RL) problem to generate code that exhibits better vectorization and locality characteristics.
- We propose a novel data augumentation strategy to extend the training set using a loop transformation which we term as *loop mutation*.
- We propose a simple and static analytical model to capture the cost of a loop by considering the instruction cost as well as number of cache misses. We use this cost model as the reward generator for the RL model.
- We integrate our RL model with the LLVM compiler so that the access and transfer of data is transparent to the end-user in an end-to-end manner.
- We evaluate the improvements obtained by our proposed method on TSVC [Bou], LLVM-Test-Suite [LOb] Microbenchmarks, PolyBench [P$^+$] and PolyBench-NN [VBPU] benchmark suites by using various metrics and statistics.

II. Background

A. Dependences and Dependence Graphs

Data and control dependences in a loop (nest) predominantly influence the parallelizability of the loop by enforcing constraints on the order of computations. Loop transformations for extracting parallelism alter the order of computations while respecting these dependences. A statement T is dependent on another statement S ($S \rightarrow T$), if there exists a dependency $S(I) \Rightarrow T(J)$ for some value of I and J in the iteration vector of the loops surrounding the statements S and T respectively.

Classically, dependences can either be *data* dependences or *control* dependences. Data dependences that impose constraints on the order of execution involve atleast one write operation and can be a flow dependence (read-after-write/RAW), an anti dependence (write-after-read/WAR) or an output dependence (write-after-write/WAW). For the code example shown in Fig. 1(a), $S3 \rightarrow S1$, has a loop carried true dependence where $S1$ reads the value of `a` written by the previous iteration of $S3$, and $S1 \rightarrow S3$, has an anti and output dependences where $S3$ reads the value of `a` written by $S1$ and write to the same.

Dependence analysis algorithms identify such dependences in the loop nest. And, these dependences can be represented in the form of a graph, called the dependence graph. A dependence graph can show dependences at every point of iteration or in a condensed manner across iterations. While the former is called the Expanded Dependence Graph (EDG) where every dependence of the form $S(I) \Rightarrow T(J)$ is shown, the latter is called the Reduced Dependence Graph (RDG) with dependences of the form $S \rightarrow T$ without an explicit description of dependences across iterations [DRV00].

RDG of a loop nest represents all constraints by a smaller, generally cyclic, graph, where each vertex corresponds to

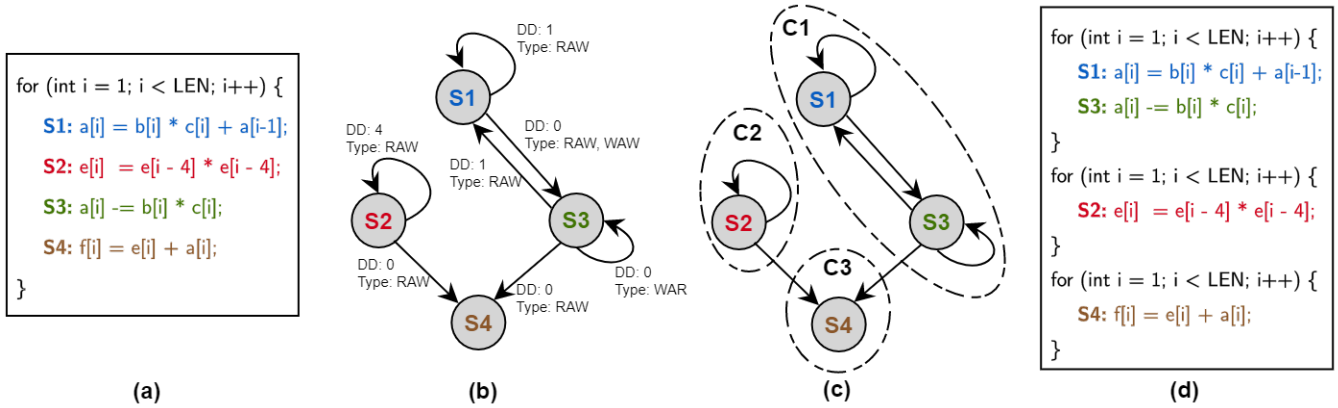

Fig. 1. Illustration of loop distribution: (a) Source Code, (b) Corresponding Reduced Dependence Graph (RDG) for the source code, (c) SCC Dependence graph (SDG) formed by identifying Strongly Connected Components in RDG, and (d) Distributed loops generated with respect to each node of the SDG (it shows maximal loop distribution).

a generic task, and the edges are labeled by dependence distances [KKP+81] [DRV00]. Fig. 1(b) shows the RDG for a given program.

B. Loop Distribution

Loop distribution or fission is a loop transformation that involves breaking a single loop into multiple loops without violating the dependence constraints. A loop `L` containing two statements `S1` and `S2` can be distributed into two loops `L1` and `L2` containing `S1` and `S2` respectively, if $S2(J) \Rightarrow S1(I)$ is not encountered or there is no dependence of the form $S2 \rightarrow S1$ across any iterations of `L`.

This transformation helps is isolating the statements with *offending* dependences, thereby exposing opportunities for better optimizations including parallelism and data locality. Allen-Kennedy proposed an algorithm for maximal distribution for identifying maximal set of completely parallel loops [ACK87].

The algorithm for identifying the set of maximal distribution involves three steps.

1) Computing an acyclic graph from strongly connected components (SCC) of the RDG corresponding to the input loop.
2) Obtaining a topological ordering of the components of the SDG of the RDG.
3) Creating new loops corresponding to each component of the SDG in the order obtained in the previous step by duplicating the original loop structure.

For the example shown in Fig. 1, there are three strongly connected components: `C1={S1, S3}; C2={S2}` and `C3={S4}`, with a topological order of $C1 \preceq C2 \prec C3$. Consecutively, three loops containing the statements corresponding to each of the components that obeys the topological ordering can be created as shown in the Fig. 1(d). Such a transformation is sound as it satisfies the dependence constraints by construction. As a result of this transformation, we have exposed a completely parallelizable loop containing the component $C3$; loop with $C2$ becomes vectorizable and another loop exposes better locality than earlier, as against the input loop containing offending dependences that was not vectorizable/parallelizable.

However, it can be understood that the maximal distribution does not always ensure optimal performance. Determining the order of distribution and which portions of the loop to be distributed is a non trivial NP-Hard problem. So, in this work we try to solve the loop distribution problem by using a reinforcement learning approach that results in semantically correct transformations by imitating the Allen-Kennedy's maximal loop distribution algorithm.

C. Reinforcement Learning

Reinforcement learning [SB18] is a branch of machine learning which involves learning by *experience*, as opposed to learning by ground truth. Here, an *agent* determines the *action* to be performed on a given *state* of the input. The final outcome depends on the sequence of actions performed on the input *observation* from the *environment*. Every action results in a change in the state of the observation. Depending on how this change impacts the environment — positively or negatively — with respect to the outcome, a *reward* is given to the agent. The agent learns to improve the prediction of actions in order to maximize the reward. Reinforcement learning does not require a *ground truth*. Also, reinforcement learning models are usually Markov Decision Processes (MDP), where the current action depends only on the current state of the input, and not on the previous history.

In this work, we model loop distribution as a Reinforcement Learning (RL) problem as there is no single correct solution, and the solution space can potentially be very large. Hence enumerating them for forming ground truth is not ideal. Also, RL modelling supports adding constraints on the action space of the model to ensure the correctness of transformation. We explain the modelling setup of our problem in detail in Sec. III-C.

D. Program representations

Any machine learning model needs inputs to be represented in a mathematical form – as vectors or matrices. These representations can either be discrete or continuous. The discrete valued representations are often feature based representations, where the features are extracted by domain experts. Contrarily, continuous valued representations called embeddings are learnt. These embeddings eliminate the need for feature extraction, and they automatically learn better description of the input data. word2vec like embeddings in natural language processing is one such prominent example. Earlier works on using machine learning for the applications in source code, have used feature based representations [FMT+08], [GWO13], [MDO14], or used the natural language processing approaches on programs [ABLY19], [FGT+20]. However, in recent times, several works [VAJ+20], [BNJH18], [SCZW20] have been proposed to learn better representations for source codes as continuous vectors, that can be used as input for machine learning models.

IR2Vec [VAJ+20] is one such work that learns program embeddings in LLVM IR as distributed vectors by considering syntactic and semantic information of the program. In specific, IR2Vec builds the representation of the program or function starting from the instructions of LLVM IR, and hence can be useful in representing an arbitrary region of the program. Initially, IR2Vec learns the vocabulary containing the representations of opcodes, types and operands of the LLVM IR. This vocabulary is used to generate representations at function level or program level. Additionally, IR2Vec also considers program information like *use-def*, *live variable* and *reaching definitions* to arrive at the final embeddings. These embeddings are also application independent, and has shown to be effective in various applications including identification of algorithms [VAD+22] and compiler optimizations including heterogeneous device mapping, prediction of thread coarsening factors, register allocation [VJA+] and phase ordering [JAVU22] applications.

In this work, we adapt IR2Vec embeddings for representing the loops in a program. We explain in detail about this in Sec. II-D.

E. Gated Graph Neural Networks

Gated Graph Neural Networks are commonly used machine learning model to process the data that are in the form of graphs. We use Gated Graph Neural Network (GGNN) [LTBZ16] which implements a gating mechanism using Gated Recurrent Units (GRUs) for better learning.

The learning in GGNN happens by updating the node representation. To update the node, message passing mechanism is used. A node which shares an edge with a given node is said to be connected or adjacent node. Each node has its own vector representation. To update a given node, it's own representation and adjacent nodes representation are used. Aggregation or summation of adjacent node's vector is performed. The aggregated value and node representation are passed to a transformation function which update the value of the node. The transformation function depend upon the use case of the problem. The process of updating the node value is known as forward propagation. GGNNs exploit local graph structures through graph convolutions.

III. Methodology

In this section, we explain our methodology to perform machine learning assisted loop distribution for obtaining code that exhibits better vectorization and locality characteristics. We first give an overview of our approach. Then we explain how we make use of the dependence graphs to perform distribution using GGNN and reinforcement learning models.

A. Overview

The overview of our proposed approach is shown in Fig. 2. First, we generate reduced dependence graphs (RDG) for loops of interest in LLVM IR. Next, the strongly connected components (SDG) are obtained from the RDG. This represents the step 2 of Fig. 2. Each node of this SDG can potentially form a loop on maximal distribution as explained in Sec. III-B.

SDGs form the input to the machine learning model that predicts the distribution sequence. We use IR2Vec embeddings to represent the nodes of this graph (step 3 of Fig. 2) which consist of source level code lowered to LLVM-IR. We design a reinforcement learning model to perform a topological walk on SDG, mimicking the maximal distribution, while taking decisions to either merge the current node with the previous node or distribute it. We use GGNNs to capture modifications to the graph made as a result of RL iterations and to obtain the global summary of the graph in the current state. Sec. III-C explains different components of the Reinforcement Learning model and how this model generates the distribution sequences. Finally, the predicted distribution sequence is applied and the code is transformed. Integration of our model with LLVM is explained in Sec. III-D.

B. Generating RDGs in LLVM IR

Just like the traditional loop distribution, we too use dependence graphs of the program to perform distribution, but assisted by machine learning models. A dependence graph is used to represent the loop.

For analyzing the dependences among the loop instructions we generate a reduced dependence graph (RDG) as explained earlier, but on the LLVM IR. In addition to the dependence edges, the constraint posed by the use-def relation of variables should also be respected. So, we account *def-use* edges along with the *memory dependence* edges. Def-Use edges track the uses of a particular definition of a variable in the program. Dependence edges are computed for each of the unordered dependences (WAW, WAR and RAW memory accesses). To compute the RDGs for distribution in LLVM IR, we create the program slices to represent a high-level statement corresponding to the nodes of RDG. These nodes are later annotated with the def-use and memory dependence edges to form RDG. Once RDGs are created, SDG is computed as explained in Sec. II.

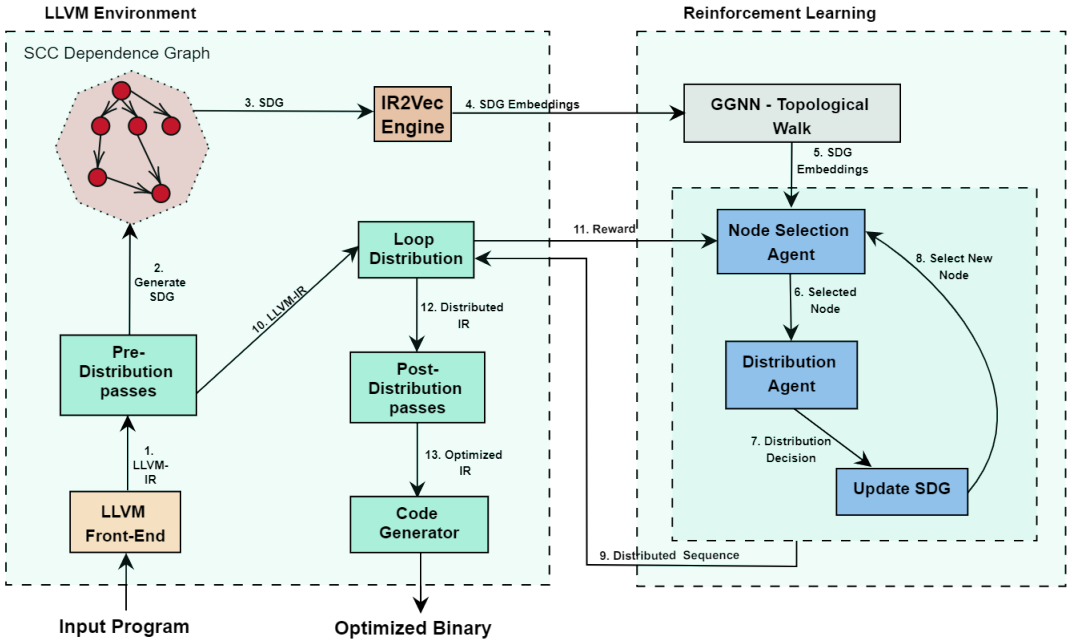

Fig. 2. Overview of the proposed approach

Initially, each instruction from LLVM-IR forms a node in the dependence graph. After analyzing the def-use dependences among instructions, we create def-use edges for the corresponding nodes. Then we create the program slices for each store node. Each such store slice corresponds to a statement in the program source and becomes the new supernode with all the corresponding LLVM-IR instructions as part of it. Since this graph can have disconnected components (disjoint graph), we create a root node and connect it with each of the sliced components. This makes every supernode of the graph reachable from the root node. Now we compute memory dependences present inside loop and create weighted dependence edge between corresponding sliced supernodes, where weight represents dependence distance.

1) Program Slices for RDG Nodes: As mentioned earlier, the statements in the high level program are decomposed into multiple instructions in the LLVM-IR. Since dependence edges involve at least one write access, we primarily focus on store instructions in LLVM IR. Each node in the generated RDG consists of the instructions starting from the last definition of the variable up till its next definition.

For this process, we obtain the program slices starting from a store instruction in a bottom-up fashion to represent an RDG node in LLVM IR. Program slices are obtained by recursively tracing the definitions of all the variables involved in an instruction, starting from the store node. This process results in the number of RDG nodes equal to the number of store LLVM-IR nodes in the loop.

The main idea behind this is to consider set of all instructions that can potentially form a semantically equivalent loop. Since LLVM IR by design is in SSA form, this process of obtaining program slices is highly simplified.

The program slices from the store instructions just represent the collection of instructions that take part in an assignment statement corresponding to the high level code. Hence the RDG generated in the way would closely resemble the RDG of the high level code. Such a way of representing the RDG would help us in distributing the loop more elegantly as shown in the Sec. III-B2. An example is shown in Fig. 3. The control flow graph with LLVM-IR for the loop from Fig. 1 is shown in Fig. 3(a). The IR instructions corresponding to the different statements of the loop are shown with different colors. Fig. 3(b) shows the corresponding program slices (PS1, PS2, PS3 and PS4) for the 4 statements (S1, S2, S3 and S4).

IR instructions that are not part of any of the store nodes are added into the RDG as separate nodes so as to preserve the dependences of the program. We merge these nodes if possible. As we maintain the edges for every node in the beginning, on creation of RDG, the incoming and outgoing edges for these nodes are adjusted and preserved accordingly. This helps loop distribution to preserve order and dependences. We show this in Fig. 4.

2) Strongly Connected Components: Nodes in the SDG are created by merging the nodes in the RDG corresponding to each components of the SCC. The edges of the nodes that lie within the SCC are removed, and those edges across the SCC are retained. By construction, each SCC node of this SDG can form an independent loop and hence can be distributed. An example SDG is shown in the Fig. 4. Here the program slices $PS1$ and $PS3$ together form a strongly connected component $C1$, and other slices form $C2$ and $C3$. Fig. 1(a) has 3 SCC nodes from 4 statements of the source code. Fig. 4 Shows the SDG for the LLVM-IR of the same loop shown in Fig. 3(a).

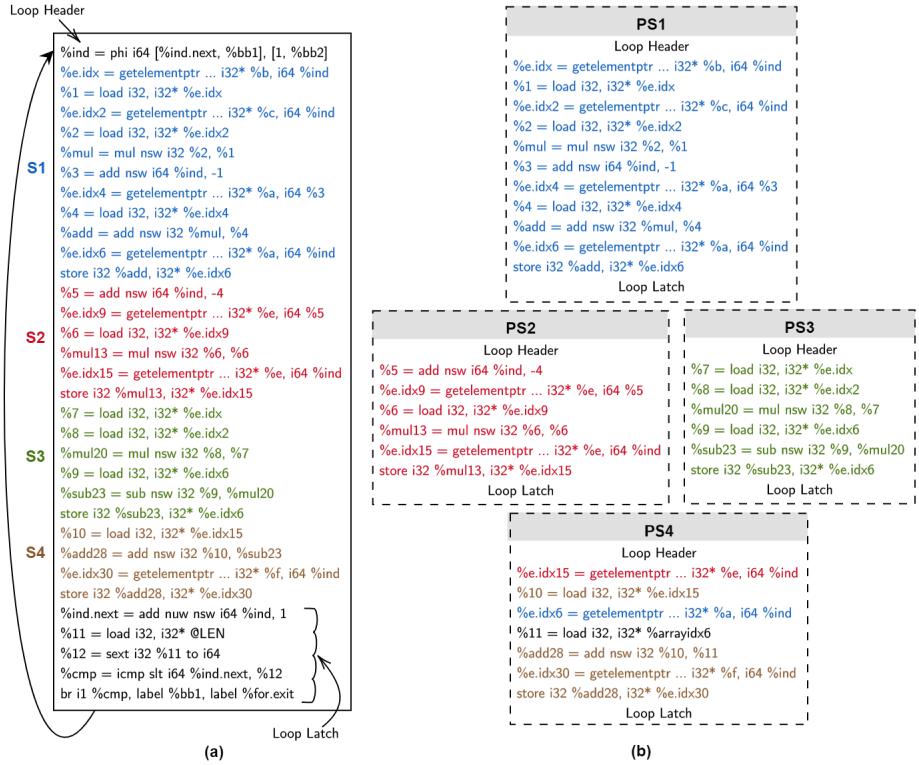

Fig. 3. (a) IR for the Source Code (b) Program Slices generated from the IR; Each program slice represents each statement of the Source Code

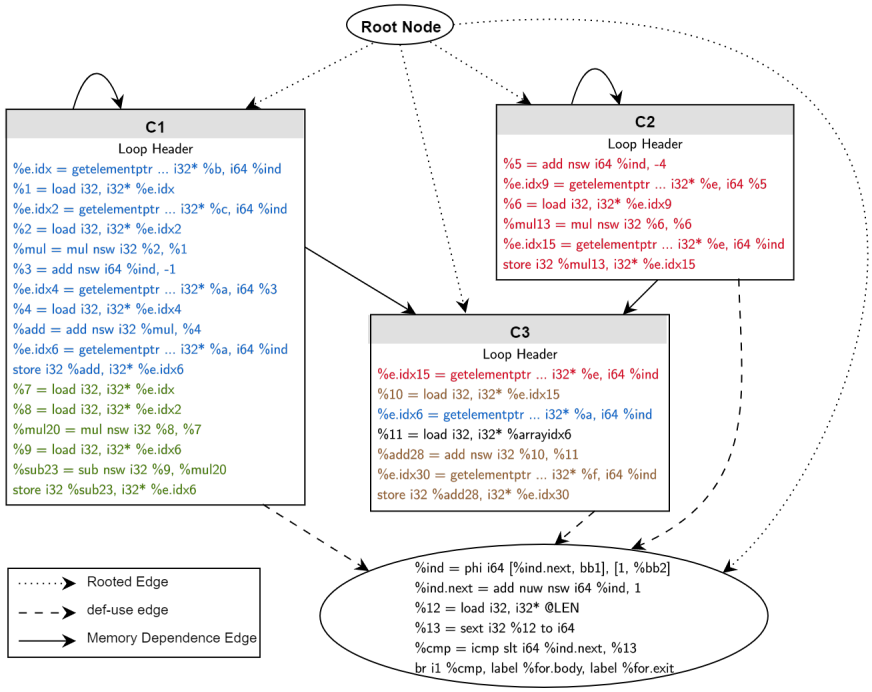

Fig. 4. SCC Dependence Graph (SDG)

C. Model

The generated graphs are used as inputs in our modelling pipeline as shown in step 2-4 of Fig. 2. In this section we explain how we represent SDGs using IR2Vec representations and the way we model reinforcement learning model to consume this information using Gated Graph Neural Networks. We also discuss various parameters of the model, including the design of our cost model, with which we arrive at a relative measure of time that takes into account of both locality and vectorization.

We implement our machine learning framework using multiagent RL model with two agents, *Node Selection Agent* and *Distribution Task Agent*. *Node Selection Agent* selects one node from the set of eligible/legal nodes. *Distribution Task Agent* chooses either to distribute or merge two nodes.

1) State: Representations from GGNN: Fig. 2 shows the workflow of the reinforcement learning model for our problem. The model takes SDG as input (step 4 of Fig. 2) and maps it to a GGNN. Initial representations of the nodes in SDG graph are generated using the embeddings obtained from the IR2Vec framework. For this purpose, each instruction in the node is represented using the n-dimensional instruction vectors in $\mathbb{R}^n$, and hence a node with m instructions is represented as $\mathbb{R}^{m \times n}$.

This graph representation forms the initial input to the GGNN model, which then processes this input to generate the final representation emebedding vectors for each node. In addition to the embeddings, each node contains the annotation vector that describes the state of the node. Specifically, we use two bits as annotation – one to denote if a node is visited, and another to show if the node is a start node. This information is then propagated to the neighboring nodes using the message passing mechanism across time as in characteristic of GNNs. This final representation is used to represent the state of the environment (step 5 of Fig. 2) for the reinforcement learning model.

We use the propagation module of the GGNN to get the updated hidden state vectors at any instance of time. Hence, the initial representation h of each node n is given as $h_n = [\![I]\!]$; where $[\![I]\!]$ denotes the embeddings obtained by IR2Vec. The hidden representation H at any instant t of n is the learned representation obtained by using the concatenation of h_n with the annotation vector of the node a_n as input to a fully connected layer of a neural network. It is given by $H_n^t : NN([h_n^t : a_n^t])$. Where $a_n = [isVisited, isStartNode]$. While processing each SDG, in the beginning, annotation vector is [1, 0] for all the nodes as no node has been visited yet. With every iteration of message passing, the representation of n is updated as $h_n^{t+1} : GRU(H_n^t, x_n)$, where GRU denotes the Gated Recurrent Unit, and $x_n = \sum_{n'} H_{n'}^t,\ where\ n' \in neigh(n)$. We obtain the state of the nodes at the end of the message passing process. The set of all possible reachable nodes at the start of the topological walk forms the initial state of the environment and is given as $state_0$. Given the initial state, the agent makes a decision on which node to visit next.

On visiting a node, the corresponding annotation vector is updated by marking *isVisited* field to True. In case of merge, we update the GGNN structure by inserting a new edge called 'Pair Edge' between the source and visited node. Later, we perform propagation on the GGNN to update the hidden state of the node. We took the summation of the vector adjacent's nodes and compute x. The GRU function take the current hidden state H_n^t and x to compute the new hidden state h_n^{t+1}. This process is continued until all the nodes are visited.

2) Agents: Node Selection Agent selects one node from the set of eligible/legal nodes and the *Distribution Task Agent* chooses either to distribute or merge two nodes.

Node Selection Agent The node selection agent picks one of the legal vertices that can be visited next in the topological order of the input dependence graph (step 6 of Fig. 2). This ordering preserves the legality of the loop distribution. The node representations H from the GGNN forms the input to this agent. The action space of the agent is constrained by using the action mask to the possible set of nodes that can be visited next in the topological order.

Distribution Task Agent *Distribution Task Agent* determines if the node chosen by the node selection agent has to be merged or distributed with the node that was selected in the previous step. Essentially this agent maintains a list of nodes to be merged, if the decision is taken to merge; and distributed otherwise. It can be seen that this process of merge or distribute with the previous node permits merging multiple nodes (step 7-8 of Fig. 2), if the decision taken on the node in the previous step was to merge. For instance, if the decision was taken to merge a node C_x with another node C_y in the previous step ($C_x \cup C_y$), then on making a decision to merge the current node C_z with the previous node would yield a merge of three nodes $C_x \cup C_y \cup C_z$. And, in the next step for a node C_p, if a decision was taken to distribute, the previous set of merge components consisting of C_x, C_y, C_z would form a separate loop and C_p marks the beginning of a new loop with which another component/node can be potentially merged based on the decisions in the future steps. Hence the distribution decision in a way marks the *boundary* between previous and current loop components.

In order to make these decisions, this agent considers the embeddings of the two nodes under consideration as the observation. The sequence of decisions made by this agent in an episode is collected as the nodes to be merged and distributed and is communicated back to the environment for code generation (step 9 of Fig. 2). The decisions made by this agent is always legal, as each node is a strongly connected component essentially ordered topologically.

3) Action: Loop Distribution: The predicted sequence is passed on to the compiler for code generation. The compiler then generates the code by creating new loops based on the decision to merge or distribute. Initially, all the statements corresponding to each component along with the other instructions that can form a loop are collected, as explained in Sec. II. The union of set of instructions/statements from the different components that are marked to be merged together across the distribute boundaries are obtained. New loops are created with these union of components. For the example

shown in Fig. 4, if $(C1 \cup C3), C2$ was the sequence generated by the agent, where $\cup$ marks the merge decision and comma marks the distribution boundary, two loops are created with one containing the statements from $C1$ and $C3$, and the other containing the statements from $C2$. Such a distribution would still preserve semantics of the program, as the distribution algorithm mimics the maximal loop distribution by respecting the dependences of the program.

4) Reward: Loop cost: For every action, the environment returns a reward (step 11 of Fig. 2) signifying the goodness of the predicted action. We use a static cost model to calculate the reward based on the cost of the loop at the end of the topological walk on the SDG. A negative reward signifies that the distributed loop is not efficient. We learn to maximize the total reward in order to obtain better loop distributions.

As mentioned earlier, loop distribution can isolate obstructing dependences from the loop which helps them get vectorized. On the other hand, cache locality of a loop can either be improved or disrupted by distribution. Worse cache locality may incur penalties that may offset the improvements obtained from vectorization. We capture these information in the form of a loop cost and use it as the reward for our model. We calculate the LoopCost as the sum of vector/scalar instructions cost (IC) and locality cost (LC) given in eq 1.

$$LoopCost = IC + LC \tag{1}$$

Here, IC captures the cost of arithmetic instructions and LC captures the cost of Memory Instructions. We obtain the instruction costs ($InstCost$) from the instruction cost tables of LLVM from the `TargetTransformInfo` pass. IC is computed as the product of the cost of non-memory instructions in the loop and the number of times the instruction would execute. The number of executions can be obtained from the trip count of the loop (TC) and we fixed it to some constant if not known at compile time. Hence, IC is calculated as:

$$IC = \frac{\sum_{I \in NonMemoryInsts} InstCost(I) * TC}{(VF * IF)} \tag{2}$$

While training, we compute the loop cost of the original loop before applying the distribution decisions and again after applying distribution and vectorization passes. We use the vector width of the loop (VF) along with the Interleave factor (IF) after vectorization in the computation of IC in Eqn. 2. VF and IF values are set to 1, if the loop is not vectorized and is in scalar form. This helps in assigning a lower cost to an instruction based on the vector factor in comparison to an instruction which is not vectorized. This also helps in quantifying if the distribution enabled vectorization.

The locality cost is computed based on the number of cache hits and misses. We create a simple analytical cache modelling to approximate the number of cache hits/misses. For this, we construct an undirected adjacency graph $G(V, E)$, where V is the set of memory read/write instructions in the loop and edges E form the data dependencies between them with a dependence distance lesser than a threshold. These edges are filtered based on a threshold which directly correspond to the temporal locality of the accesses. Additionally, we add edges between the memory instructions that share the same base pointers. These accesses are also filtered based on a manually selected threshold to determine if the accesses are spatially close, corresponding to the measure of spatial locality. Hence, the data dependence edges correspond to temporal cache hits and spatial edges correspond to spatial cache hits.

```
for (int i = 1; i < LEN1D; i++)
{
    a[i] = a[i-1] + c[i] * d[i];
    b[i] = b[i+1] - e[i] * d[i];
}
```

```
A: %0 = load a, i-1
B: %1 = load c, i
C: %2 = load d, i
....
D: store %x, a, i
E: %3 = load b, i+1
F: %4 = load e, i
...
G: store %y, b, i
```

Fig. 5. Example loop and it's psuedo IR

An edge in Graph G represents a dependence, if the dependence distance is small then regardless of the direction of the edge one of the access will be a cache miss and other will be a cache hit. Suppose (u, v) is a dependence edge then all accesses of u are compulsory misses and subsequently v's accesses are all hits because they are accessing a common memory item which will be present in cache by the time v executes.

By definition, minimal vertex cover of graph G contains either vertex u or v in its set. All independent vertices of the graph indicates memory accesses that are not aliasing with any other memory space, hence they all will incur cache misses. So the set containing minimal vertex cover plus all the independent vertices forms memory instruction that will incur cache misses.

Fig. 5 shows the example code and it's Psuedo IR. Here, the set of edges for the graph are {(A, D), (E, G)}. For the two edges statement A and E will incur cache misses and statements D and G will have cache hits. Minimal vertex cover for this graph is {A, E} or {D, G}. Statements B, C, and F are independent vertices and they too cause cache misses. The final set contains {A, E, B, C, F}.

We compute a set (MS) with the minimal vertex cover of the graph G plus the independent vertices. The memory accesses instructions that belong to the set MS will incur all the compulsory misses in the loop and the remaining accesses are considered as cache hits.

NCM is the number of cache misses as in eq. 3. Where $TC * Stride * DataSize$ is the length of the array in bytes that is accessed in the loop, which is divided by cache line size (CLS) to give the number of compulsory cache misses.

$$NCM = \frac{\sum_{MI \in MS} TC * Stride * Datasize(MI)}{CLS} \tag{3}$$

We compute the total number of memory accesses in the loop (TMA) as shown in eq. 4. Here, NumMemInsts is the number of memory instruction in the loop.

$$TMA = NumMemInsts * VF * IF * TC \tag{4}$$

Total number of cache hits is shown in eq. 5 is the difference between TMA and NCM.

$$NCH = TMA - NCM \tag{5}$$

Accesses causing cache misses will have higher cost than cache hits so, we approximate total cache cost by adding extra weight to cache misses shown in eq 6.

$$\begin{aligned} LC = &\alpha * MemoryInstCost * NCM + \\ &(1-\alpha) * MemoryInstCost * NCH \end{aligned} \tag{6}$$

For our experiments we have set α to 0.7. The above cache cost computation does not take into consideration conflict, capacity and other types of cache misses. It serves as good approximation as a reward for training our model.

D. Integration with LLVM compiler

We integrate the Python based Reinforcement Learning (RL) model with the LLVM compiler by using C++ Python bindings. We added a transformation pass in LLVM to perform ML guided loop distribution and scheduled it in place of the original loop-distribution pass in the O3 pipeline. Hence, on invoking this modified O3 pipeline, first all the passes that are scheduled before the distribution pass (Pre-Distribution passes) are invoked. Then the ML assisted loop distribution is performed.

This pass iterates over all functions and collects the loops for which RDG can be generated. On generating the RDGs, the corresponding SDGs are generated and then we compute the corresponding IR2Vec embeddings. The embeddings along with the list of loops are passed to the RL model using C-Python bindings. For training, the model learns the distribution sequence and passes it to LLVM to obtain the reward in the form of loop cost. On receiving the distribution sequences from the model, our pass would proceed to distribute the loop as per the predictions. This process is continued for all the loops, for each function. After loop distribution, the remaining passes of the O3 sequence (Post-Distribution passes) are applied to generate the final binary. In case of inference, the model predicts the distribution sequence for a loop and communicates it back to the LLVM environment.

IV. Experimentation and Results

A. Train/Test Dataset

We train the model by generating SDGs for the inner-most loops. As loop distribution does not distribute loops with function call(s) and/or conditional statements inside it, we do not generate SDGs for these cases. We also ignore the loops having multiple exit blocks or reduction phi nodes. Reduction phi nodes help in reducing number of loads and stores. We ignore the loops having reduction phi nodes because of complexity to handle this.

On applying these conditions, we were able to generate only 21 SDGs for TSVC benchmark (152 loops). However, for training a Reinforcement Learning model we require good number of datapoints. Hence, we propose a technique called *loop mutation* to generate more number of loops given the seed set that satisfy the desired properties. Loop mutation is a transformation that changes the dependence distances of a loop, given a SDG and generate the new loops (SDGs). On performing loop mutation, we generate 260 SDGs in total from the TSVC benchmark. Even though this is the small set of data to train a reinforcement learning model, we are able to cover loop patterns from the standard benchmarks.

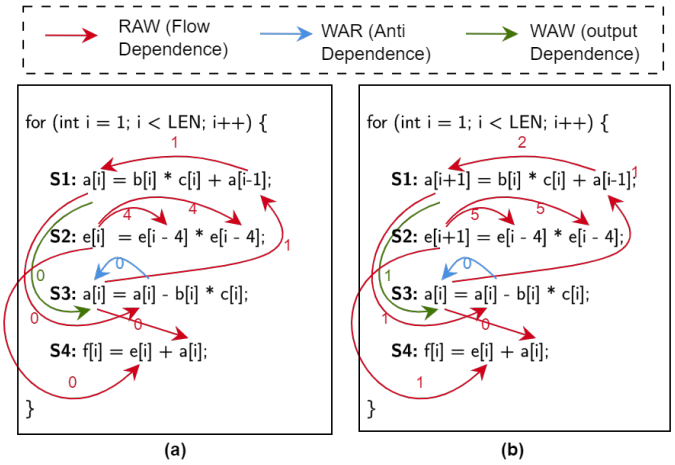

Fig. 6. (a) Loop and its dependences before mutation (b) Loop and its dependences after mutation with distance 1

1) Data generation: Mutation: Loop-mutation is a loop transformation that changes the dependences between instructions and produces a new loop, while the instructions and order between them remains the same. The process of loop mutation involves changing the dependence distances of the accesses in the loop from $k1$ to $k2$, such that $k2 = k1 + \beta$, where we set $\beta \in [-5, 5]$. In the process of loop mutation, if we mutate both the read and write accesses of a variable with the same constant, the effective dependence distance may not change. Hence, we mutate only the write accesses, leaving the read accesses intact for the flow and anti dependences. For output dependence, we randomly mutate any one write access.

For the earlier example from Fig. 1, we show the dependence distances for each array access in Fig. 6(a). Fig. 6(b) shows the resultant loop on performing loop mutation by setting the value of $\beta = 1$. The read accesses involved in flow dependences - a[i] in S1 and e[i] in S2 are mutated to a[i+1] and e[i+1] respectively, thereby changing the corresponding dependence distances. This mutation also results in changing the dependence distance corresponding to the output dependence between a[i] in S1 and S3.

We perform mutation over loops from TSVC benchmark and generate 260 loops, which is our dataset for training RL model.

B. Configuration

We use LLVM-10 for all our experiments. We use DQN Algorithm [HGS16] to perform training on the 260 loops generated from TSVC benchmark.

We use diferent set of benchmarks Microbenchmarks from LLVM-Test-Suite [LOb], PolyBench [P$^+$] and PolyBench-NN benchmark for validation.

We set maximum number of nodes in the GGNN to 1000, with these nodes being connected with a dense layer. Our neural network model consist of 3 dense layers with ReLU as the activation function. We use Adam optimizator [KB15] and set learning rate to 5e-4.

Our proposed model starts training in batches with the batch size of 64 with replay buffer size of 10,000. To introduce exploration in training, we gradually decrease the exploration rate with time. We categorize the `O3` pass sequence into 3 parts: Pre-DistributionPasses, loop-distribution and Post-DistributionPasses, and replace the original loop distribution of O3 with our modified loop distribution pass.

We train our model on Intel Xeon E5-2697 processor with 36 cores and 2 threads per core. It took around 8 hours to train the mentioned model.

C. Results

We evaluated the performance of our framework by validating the programs from TSVC [Bou], LLVM-Test-Suite Microbenchmarks [LOb], PolyBench [P$^+$] and PolyBench-NN [VBPU] benchmarks. For all our experiments baseline is `O3`. For the set of benchmarks that we consider, loop distribution is applied as predicted by the model. In order to mimic the flow of `O3` optimization sequence, the Pre-Distribution pass sequence is run before the distribution; the latter is followed by the Post-Distribution passes. Using this optimized IR, we generate the binaries and compute the execution time by running them 3 times and taking their average. We compare this execution time with that generated with the standard `O3` optimization sequence.

In Fig. 7(a), we show the execution time improvement on applying `O3` with loop distribution suggested by our framework in comparison to that of `O3` pass sequence. We observe that our framework achieves a maximum improvement of 4.61% for LLVM-Test-Suite. Similarly it achieves 3.62% improvement for gramschmidt benchmark from PolyBench and 3.80% for sumpool from PolyBench-NN. While, on heat-3d and CNN, there is a small degradation.

In Fig. 7(b), we also show the percentage improvement using *executed Instructions Per Cycle* (IPC) as another metric with our flow. We observe that for gramschmidt, IPC is similar to O3. Also, for microbenchmarks from llvm-test-suite the IPC is less than O3. For all the other benchmarks, the IPC of our flow is improved. We observe that we achieve maximum improvements in IPC for the PolyBench-NN benchmarks i.e. CNN, MaxPool, SumPool benchmarks. We observe that average improvement in IPC is 7.10% for tsvc, 3% for PolyBench and 14.29% for PolyBench-NN benchmarks.

In Table II, we show the number of loops distributed in each benchmark. For all our experiments, we show only the benchmarks for which there is at least one successful loop distribution, and skip the remaining benchmarks from the benchmark-suite. We also compare some of the metrics

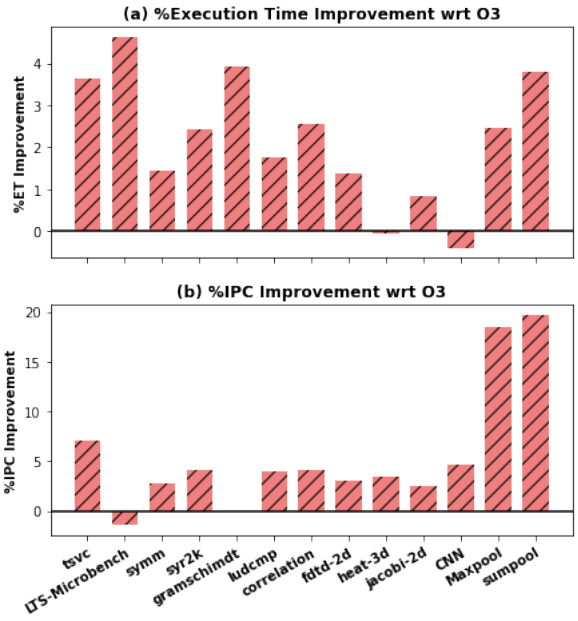

Fig. 7. Percentage of Execution Time and IPC improvement wrt O3

TABLE II
Statistics- #LD: number of distributed loop, #LV: number of loop-vectorization

		#LD	#LV		L1 Hit %	
		-O3 -cld	**-O3**	**-O3 -cld**	**-O3**	**-O3 -cld**
TSVC	**tsvc**	21	662	**684**	60.0	**65.7**
LTS	**Micro-benchmarks**	15	222	**265**	58.9	**60.3**
PolyBench	**symm**	1	5	5	31.5	**31.6**
	syr2k	1	5	5	9.1	9.1
	gramschimdt	1	4	**5**	9.0	**20.4**
	ludcmp	1	11	**12**	10.1	**10.3**
	correlation	1	5	**6**	8.7	**30.9**
	fdtd-2d	1	9	**11**	**21.0**	20.5
	heat-3d	1	6	**7**	40.5	**41.4**
	jacobi-2d	1	6	**7**	35.6	**36.0**
PolyBench-NN	**CNN**	3	4	4	44.3	**45.7**
	Maxpool	2	4	4	33.7	**36.4**
	sumpool	3	4	4	33.4	**36.5**

(number of loops vectorized and percentage of L1 cache hits) with our flow (-O3 -cld) and with O3 optimization sequence. We observe that the number of loop vectorization is improved for tsvc, LLVM-Test-suite and PolyBench benchmarks. We also compare L1 cache hits and found that our flow is better utilizing the cache as percentage of cache hits is improved.

V. Related Works

Earlier, several approaches have been proposed to improve parallelism and/or data locality by performing better loop distribution and loop fusion [ACK87], [KM94], [LZS$^+$05]. Also there have been many works in order to improve the

performance using both SLP and loop vectorization. Alexandre et. al. [EWO04] proposed to vectorize loop having misaligned memory refernces. Xinmin et. al. [TSS+16] presents the LLVM intrinsic functions for parallelization, vectorization and offloading.

The current LLVM heuristic for loop distribution provides support only for inner-most loops. The loop distribution algorithm in LLVM is designed primarily as an enabler for vectorization [KF18]. However, the distribution pass is disabled in the default pipeline for all optimization sequences.

In recent times, machine learning based approaches to solve compiler optimization problems have been applied for inlining [KCWS13], vectorization [SPS12] [HAAW+20], unrolling [SA05], register allocation [VJA+] and phase ordering [JAVU22] [HHAM+19] etc. To the best of our knowledge ours is the first framework which tries to optimize loop-distribution using machine learning in order to achieve better vectorization and improvement of locality.

There are various works that try to solve the compiler optimization based problems using deep reinforcement learning approach. Mendis et al. [MYP+19] propose a Superword Level Parallelism (SLP) vectorization scheme which is better than the LLVM compiler heuristics. For instruction packing problem, it imitates the optimal decisions made by an ILP solver proposed in goSLP [MA18]. This work uses graph neural network policy where Instructions are represented as a connected graph. Neurovectorizer, proposed by Ameer Haj-Ali et al. [HAAW+20] is a framework for Vectorization that integrate Deep Reinforcement Learning with the LLVM compiler. Training dataset for this framework are simple synthetic loops. These loops are fed to Code2Vec based embedding generator for representing loops. These learned embeddings are input to a Deep RL agent which determines the vectorization factor. Mircea Trofin et al. [TQB+21] propose MLGO framework to replace a heuristic based inlining for size optimizations with a (policy gradient and evolution strategies) ML model. Amir H. Ashouri et al. [AEH+22] propose MLGOPerf framework to solve LLVM's inliner for optimizaing performance using RL.

VI. Conclusion

We propose a reinforcement learning based approach to solve the loop distribution problem in order to improve loop vectorization and enhance locality. We generate SCC dependence graph(s) for loop(s) as inputs to the model. We compute IR2Vec representation of the SDGs and give it to the model. Our model computes static rewards `LoopCost` that is computed with instruction cost and locality cost. We are able to obtain improved vectorization and cache usage with our method over O3 optimization pipeline of LLVM-10. We show improvements in execution time and instruction per cycle for TSVC, LTS-Microbenchmarks, PolyBench and PolyBench-NN benchmarks over x86 architecture.

In future, we plan to extend this work by performing more inlining to get more explosure to the loop-distribution. We also plan to experiment this work on more and bigger datasets like SPEC.

Acknowledgement

We are grateful to Venugopal Raghavan and AMD compiler team for insightful discussions at the early stage of this work.

This research is funded by the Department of Electronics & Information Technology and the Ministry of Communications & Information Technology, Government of India. This work is partially supported by a Visvesvaraya PhD Scheme under the MEITY, GoI (PhD-MLA/04(02)/2015-16), an NSM research grant (MeitY/R&D/HPC/2(1)/2014), a Visvesvaraya Young Faculty Research Fellowship from MeitY, and a Google PhD Fellowship.

References

[ABLY19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019.

[ACK87] r. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 63–76, New York, NY, USA, 1987. Association for Computing Machinery.

[AEH+22] Amir H. Ashouri, Mostafa Elhoushi, Yuzhe Hua, Xiang Wang, Muhammad Asif Manzoor, Bryan Chan, and Yaoqing Gao. Mlgoperf: An ml guided inliner to optimize performance, 2022.

[AZLY19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[BNJH18] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 3589–3601, USA, 2018. Curran Associates Inc.

[Bou] Michael Boulton. TSVC_2. https://github.com/UoB-HPC/TSVC_2.git. Accessed 2015-09-16.

[DRV00] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and Automatic Parallelization*, volume 85. 01 2000.

[EWO04] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 82–93, New York, NY, USA, 2004. Association for Computing Machinery.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.

[FMT+08] G. Fursin, C. Miranda, O. Temam, Mircea Namolaru, Elad Yom-Tov, A. Zaks, Bilha Mendelson, Edwin V. Bonilla, J. Thomson, H. Leather, Christopher K. I. Williams, M. O'Boyle, Phil Barnard, Elton Ashton, E. Courtois, and F. Bodin. Milepost gcc: machine learning based research compiler. 2008.

[GWO13] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 22:1–22:10. IEEE Computer Society, 2013.

[HAAW+20] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. *NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning*, page 242–255. Association for Computing Machinery, New York, NY, USA, 2020.

[HGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[HHAM+19] Qijing Huang, Ameer Haj-Ali, William S. Moses, J. Xiang, I. Stoica, K. Asanović, and J. Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308, 2019.

[HLF+18] Ding-Yong Hong, Yu-Ping Liu, Sheng-Yu Fu, Jan-Jan Wu, and Wei-Chung Hsu. Improving simd parallelism via dynamic binary translation. *ACM Trans. Embed. Comput. Syst.*, 17(3), feb 2018.

[JAVU22] Shalini Jain, Yashas Andaluri, S. VenkataKeerthy, and Ramakrishna Upadrasta. Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 121–131, 2022.

[KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[KCWS13] Sameer Kulkarni, John Cavazos, Christian Wimmer, and Douglas Simon. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, 2013.

[KF18] Michael Kruse and Hal Finkel. Loop optimizations in llvm: The good, the bad, and the ugly. https://llvm.org/devmtg/2018-10/slides/Kruse-LoopTransforms.pdf, 2018.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, page 207–218, New York, NY, USA, 1981. Association for Computing Machinery.

[KM94] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 301–320, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[LOa] LLVM-Org. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html.

[LOb] LLVM-Org. LLVM Test Suite. https://github.com/llvm/llvm-test-suite. Accessed 2021-08-25.

[LTBZ16] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[LZS+05] M. Liu, Q. Zhuge, Z. Shao, C. Xue, M. Qiu, and E.H.-M. Sha. Maximum loop distribution and fusion for two-level loops considering code size. In *8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN'05)*, pages 6 pp.–, 2005.

[MA18] Charith Mendis and Saman Amarasinghe. Goslp: Globally optimized superword level parallelism framework. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[MDO14] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.

[MYP+19] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. *Compiler Auto-Vectorization with Imitation Learning*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 132–143. ACM, 2006.

[NZ08] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, page 2–11, New York, NY, USA, 2008. Association for Computing Machinery.

[P+] Louis-Noel Pouchet et al. PolyBenchC. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.

[SA05] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*, pages 123–134, 2005.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SCZW20] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. Flow2vec: Value-flow-based precise code embedding. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[SPS12] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.*, 8(4), jan 2012.

[TQB+21] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and Xinliang David Li. Mlgo: a machine learning guided compiler optimizations framework, 2021.

[TSS+16] Xinmin Tian, Hideki Saito, Ernesto Su, Abhinav Gaba, Matt Masten, Eric Garcia, and Ayal Zaks. Llvm framework and ir extensions for parallelization, simd vectorization and offloading. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 21–31, 2016.

[VAD+22] S VenkataKeerthy, Yashas Andaluri, Sayan Dey, Rinku Shah, Praveen Tammana, and Ramakrishna Upadrasta. Packet processing algorithm identification using program embeddings. 2022.

[VAJ+20] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Trans. Archit. Code Optim.*, 17(4), December 2020.

[VBPU] Hrishikesh Vaidya, Akilesh B, Abhishek Patwardhan, and Ramakrishna Upadrasta. PolyBench-NN. https://github.com/IITH-Compilers/PolyBench-NN. Accessed 2018-03-12.

[VJA+] S. VenkataKeerthy, Siddharth Jain, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. Rl4real: Reinforcement learning for register allocation.