Vectorization and Distribution with Deep Reinforcement Learning

Junwei Zhou
University of Michigan, Ann Arbor

Tan Tao
University of Michigan, Ann Arbor

Xinyi Liu
University of Michigan, Ann Arbor

Leheng Lu
University of Michigan, Ann Arbor

Luoxi Meng
University of Michigan, Ann Arbor

Abstract

LLVM's baseline models for loop optimization are heuristic-based in nature and often lead to suboptimal performance. We use Code2Vec [1] to generate code embeddings that characterize loops. With embeddings plus static features generated from LLVM pass, we train a reinforcement learning (RL) model that predicts the loop optimization factors to minimize execution time. We train the model on the loop dataset extracted from LLVM test suite and tune hyperparameters. The inference results show the effectiveness of adding distribution as an optimization factor, but the static features are too trivial to be helpful.

1 Introduction

In this project, we explore how we can leverage RL to guide loop optimization in LLVM. We observe that LLVM's cost models that predict loop optimization factors will lead to sub-optimal performance. With Code2Vec and PPO [4], it becomes possible to train an RL model to solve this problem. We use NeuroVectorizer [3], an existing framework that optimizes loop vectorization with RL, and further expand the search space by introducing *loop distribution* as a new factor. We experiment with different combinations of optimization factors and feature sets, and tune hyperparameters to achieve optimal performance.

2 Backgroud

2.1 Vectorization and Distribution

2.1.1 Intro

Modern CPU and GPU cores use single instruction, multiple data (SIMD) execution units to achieve higher performance and power efficiency. The underlying SIMD hardware is exposed via instructions such as SSE, AVX, AVX2, AVX-512 [5]. To leverage this hardware support, the required software changes are referred to as *Vectorization*. Specifically, it is the process of converting a program from a scalar implementation, which performs one instruction on one data item, to a vector implementation, which performs a single instruction on multiple data items.

Vectorization could be applied to loops that operate iteratively on arrays by changing operations on single array elements to vector operations on multiple elements. Considering the following example:

```c
int A[1024], B[1024], C[1024];
for (i = 0; i < 1024; i++)
{
   C[i] = A[i]*B[i];
}
```

Listing 1: Before Vectorization

If the CPU supports vector instructions with a width of 4, the implementation with vectorized approach could be represented as:

```c
int A[1024], B[1024], C[1024];
for (i = 0; i < 1024; i+=4)
{
   C[i:i+4] = A[i:i+4]*B[i:i+4];  // Vector
     Instruction with a vector width of 4
}
```

Listing 2: After Vectorization

However, loops are not always vectorizable due to data dependencies across multiple iterations. *Loop Distribution*, which involves splitting a single loop into multiple loops, might be able to expose better vectorization opportunities. For example, if we have the following loop:

```c
int A[N], B[N], C[N], D[N], E[N];
```

```
for (int i = 0; i < N; i++)
{
    A[i+1] = A[i] + B[i];   // S1
    C[i] = D[i] * E[i];     // S2
}
```

Listing 3: Before Distribution

This loop will be split into two loops between statements S1 and S2:

```
int A[N], B[N], C[N], D[N], E[N];
for (int i = 0; i < N; i++)
{
    A[i+1] = A[i] + B[i];   // S1
}
for (int i = 0; i < N; i++)
{
    C[i] = D[i] * E[i];     // S2
}
```

Listing 4: After Distribution

After loop distribution, now we can vectorize the loop containing S2.

2.1.2 State-of-the-Art Loop Vectorization and Distribution Techniques

The available set of vector instructions and their vector widths are processor-dependent. So, it is not practical to manually configure loop vectorization and distributions.

For loop vectorization, modern compilers like LLVM provide loop vectorizers that use a cost model to determine if the loop is safe and profitable to vectorize and select the vector width [6]. Unfortunately, the cost models estimate the cost of different instructions with pre-defined heuristics, which leads to sub-optimal results in practice [3]. Another commonly used approach is Polly [2], which utilizes the polyhedral model for loop optimization. However, the process of constructing polyhedral representations simplifies the loop vectorization decisions. So far, the major contribution of Polly is tiling and loop fusion to improve data locality.

Loop distribution is currently not enabled by default in the optimizer because it can hurt performance when a premature distribution is applied. For example, instruction-level parallelism could be reduced in Listing 4 compared to Listing 3. This conservative approach can again lead to sub-optimal performance, and meanwhile, it reveals the difficulty of making these loop optimization decisions with existing approaches.

The two key questions here are: *How to characterize the loops? How to use these characteristics to predict and make optimal loop optimization decisions?* 2.2 will answer the first question and 2.3 will answer the second.

2.2 Code2Vec

Code2Vec [1] is a neural network model that relies on Natural Language Processing (NLP) for representing snippets of code as continuous distributed vectors. A code snippet is represented as a single fixed-length code vector that is used to predict semantic properties of the snippet. The Code2Vec network architecture is shown in Figure 1.

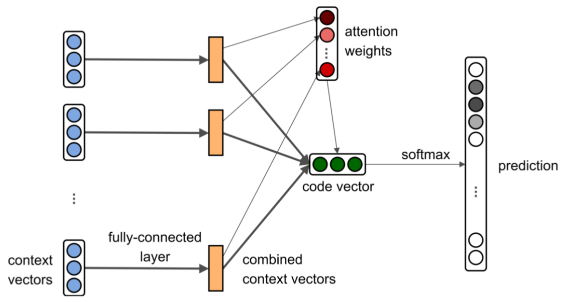

Figure 1: Architecture of Code2Vec network

The code is first decomposed into a collection of paths in its Abstract Syntax Tree (AST). An AST for a code snippet C is represented as $< N, T, X, s, \delta, \phi >$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \to (N \cup T)^*$ is a function that maps a nonterminal node to a list of its children and $\phi : T \to X$ is a function that maps a terminal node to an associated value. AST paths are defined in AST, an AST path p of length k is represented of the form: $n_1 d_1 ... n_k d_k n_{k+1}$ where $n_1, n_{k+1} \in T$ are starting terminal $start(p)$ and ending terminal $end(p)$, for $i \in [2..k] : n_i \in N$ are nonterminals and for $i \in [1..k] : d_i \in \{\uparrow, \downarrow\}$ are movement directions. An AST path is further expressed as a path-context $< x_s, p, x_t >$ where $x_s = \phi(start(p))$ and $x_t = \phi(end(p))$ are the values associated with the start and end terminals of p.

Based on the AST of code, a code snippet C is represented a set of path-contexts to be fed into the neural network. For each path-context b_i, three embeddings are concatenated to a single context vector c_i: $c_i = embedding(< x_s, p_j, x_t >) = [\text{valuevocab}_s; \text{pathvocab}_j; \text{valuevocab}_t]$ where $p_j \in P$ is a connecting path. The three embeddings are learnt from the Path-Attention Model. Then in the fully connected layer, a combined context vector is produced:

$$\tilde{c}_i = tanh(W \cdot c_i) \quad (1)$$

where W is a learnt weights matrix. Next in the attention mechanism, an attention weight α_i for each combined context vector $\tilde{c}_i$ is computed as:

$$\alpha_i = \frac{exp(\tilde{c}_i^T \cdot a)}{\sum_{j=1}^{n} exp(\tilde{c}_j^T \cdot a)} \tag{2}$$

where attention vector a is initialized randomly and learnt with the network. With the combined context vectors and attention weights, the whole code snippet is represented as a aggregated code vector:

$$v = \sum_{i=1}^{n} \alpha_i \cdot \tilde{c}_i \tag{3}$$

The prediction is preformed using the code vector to compute a dot product between the code vector v and each of the tag embeddings.

2.3 Reinforcement Learning

2.3.1 Intro

We consider a Markov Decision Process (MDP) of state S, action A, a reward function $R : S \times A \rightarrow \mathbf{R}$ and transition dynamics $P : S \times A \times S \rightarrow [0, 1]$. The reinforcement learning (RL) agent trains a policy $\pi : S \times A \rightarrow [0, 1]$ and uses reward R obtained in environment to improve the policy π.

Apart from RL, it might be possible to apply supervised learning methods to predict loop optimization factors. However, it would take a large amount of time to search for all the possible ones to find out the optimal factors, especially when the sample size is huge. That's exactly why we apply RL to solve this problem. RL agent can easily search an action space and co-optimize several objectives such as code size, compilation time and so on. And after iterations, RL agent can obtain an optimal policy for predicting loop optimization factors.

2.3.2 PPO

Proximal policy optimization [4] is an excellent method for generating consistent and easily operating policies in RL. Normal policy gradient methods can only perform one update of gradient per data sample, while PPO is capable of implementing several epochs of minibatch updates. For each step, it generates an update of the gradient to minimize the cost while providing a relatively small deviation from previous policies. PPO has a lot of benefits like trust region policy optimization, and it's even more effortless to implement with a better sample complexity in general. It is well-tested on various benchmarks such as simulated robotic locomotion and Atari game playing, and outshines other online policy gradient methods.

Here, we use this main objective function to define it as

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{4}$$

where θ denotes the policy parameter. $r_t(\theta)$ is the probability ratio under new and old policies, respectively. $\hat{E}_t$ is the empirical expectation over timestamps. $\hat{A}_t$ is an estimator of the advantage at timestamp t.The ε represents a value of hyperparameter, which is 0.1 or 0.2 in normal cases. This formula takes the minimum of both clipped and unclipped objectives, and the final objective will be the lower bound of unclipped objectives.

3 Method

The pipeline of our method is illustrated in Figure 2. The code of loops are fed into the feature construction part as the input. This part consists of code embedding generator and numerical features generator. Constructed features will be fed to the RL agent which uses PPO to generate the policy-determined vectorization and distribution pragmas. Loops with injected pragmas are compiled and executed, and the execution time along with the baseline time are used to calculate the reward of RL agent. The agent uses the reward to improve the policy and the optimal policy will be reached after iterations. In the following subsections, we are going to explain the details of each part.

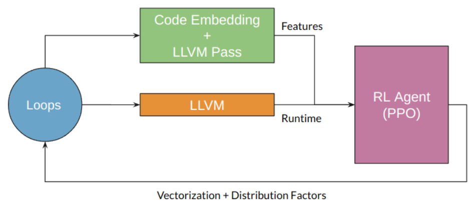

Figure 2: Pipeline of the method

3.1 Feature Construction

We generate code embeddings with Code2Vec as discussed in 2.2. Besides, we extract static numeric features through LLVM loop passes. Specifically, we will record the following features for each loop: *loop depth, number of basic blocks, number of branches, number of total instructions, number of integer instructions, number of* *floating-point instructions, number of stores*, and *number of loads*.

When there is a nested loop, we only consider the innermost loop. Then, we compare the training process with and without these numeric features.

3.2 RL Problem Formulation

To formulate this problem as a RL problem, we define (S,A,P,R) as follows.

The purpose of our method is to obtain an optimal policy to accelerate the execution of the loop code, so we use the normalized difference in the execution time as the reward. $t_{baseline}$ is the execution time when compiling with LLVM's baseline optimization, and t_{RL} is the execution time when compiling with the injected pragmas by the policy. However, because different loops' execution time vary a lot, we need to normalize $t_{baseline} - t_{RL}$ with the baseline execution time $t_{baseline}$. Therefore, we define the reward as

$$R = (t_{baseline} - t_{RL})/t_{baseline} \tag{5}$$

In addition to the execution time of the loops, compilation time also needs to be taken into consideration. To avoid the compilation taking too much time, we limit it to ten times the time of compiling with LLVM's baseline optimization. The reward is set as -9 to penalize longer compilation time.

Action A is defined as the combination of VF, IF and DF which are picked from the following values:

$$\begin{aligned} VF &\in [1,2,4,8,16,32,64] \\ IF &\in [1,2,4,8,16,32,64] \\ DF &\in [\text{"enable"}, \text{"disable"}] \end{aligned} \tag{6}$$

State S is defined as the embedding vectors and numerical features of the code. Given a state and an action, the change of the embedding vectors and numerical features should be deterministic, so the transition dynamics P of RL agent is also deterministic, which means transitioning to the next state with 100% probability.

3.3 Dataset Description

The reinforcement learning agent requires a lot of code samples for training. Studies show that code that is not restricted to loops only will slow down training due to long compilation time [3]. In addition, the number of open-source benchmarks available for training is very small. As a result, the dataset used in training the RL agent is consist of code with synthetic loops only. There are 7812 synthetic loops examples in the training set and 278 in the testing set where they differ by the names of the parameters, the stride, the number of iterations, the functionality, the instructions and the number of nested loops.

```c
//#pragma clang loop vectorize_width(VF)
    interleave_count(IF) distribute(DF)
int i;
for (i=0; i<64; i++){
    sum[i] = in1[i] +in2[i];
}
```

Listing 5: Code Example 1

```c
int i,j;
for (i = 0; i < 64; i++) {
    int result = 0;
    //#pragma clang loop vectorize_width(VF)
    interleave_count(IF) distribute(DF)
    for (j = 0; j < 2048; j+=8) {
        result += (A[i][j] *B[i][j]);
    }
    out[i] = result;
}
```

Listing 6: Code Example 2

Listing 5 shows an example of the code sample that contains a single for loop which iterates 64 times. The commented pragma line is what the RL agent is going to inject. Listing 6 shows an example of nested loop. The pragma line will be inserted before the innermost loop.

4 Evaluation

4.1 Experiments Setup

The model is evaluated based on the reward mean, which is the average reward that the model gets over batch size in each training step. A positive value means that the model is better than the baseline. We have explored two different definitions of optimization. One contains vectorization factors (VF+IF), while another contains both the vectorization factors and the distribution factor (DF). We have also explored different choices of selecting features. We have tried to train the model only with the features generated from Code2Vec, as well as adding numerical features that we collected from writing our own LLVM passes. Lastly, we tune the hyper-parameter in training to explore the best model with learning rate, the sizes of batch and hidden layer.

4.2 Results

We train two models. One only optimizes loop vectorization, the other one optimizes both loop vectorization and distribution. Figure 3 illustrates each training step's reward of the two models. After 100k steps, the latter one outperforms the former one. It implies that adding distribution optimization improves model's performance.

To get the best model, we tune hyperparameters including the size of hidden layers, learning rate, and batch size. Figure 4, 5, 6 shows that the best hyperparameters among the ones we have tried are 32x32, 5e-4, 500. We use these hyperparameters in the following experiments.

To further optimize the model's performance, we feed numerical features generated by LLVM pass to the model. Apart from directly using the vector consisting of 8 features, to make numerical features have more influence on the model's output, we also extend the numerical feature vector by copying each feature 24 times to form a vector of length 200 (8×25). We concatenate the unextended or extended vectors with the code embedding vector, and feed this feature vector into the RL agent to do the training.

Figure 7 shows that adding the numerical features decreases the model's performance, and the extended numerical features have worse influence on the performance. The reason for this might be the numerical features we generate don't contribute to the optimization, but adding more distractions to the model harms the model's performance.

The model's performance on test set is very similar to that on training set, which is illustrated by Figure 8. Compared to the model only optimizing vectorization, adding distribution optimization improves the performance, but feeding numerical features into the model harms the performance.

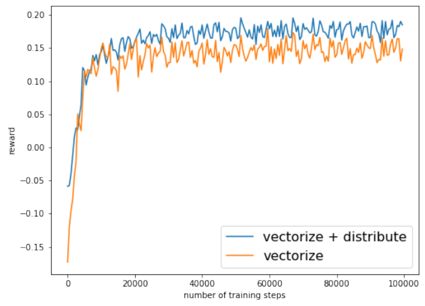

Figure 3: Reward of training

5 Related Work

Compilers translate programming languages written by humans into binary executable by computer hardware. Machine learning is a field of artificial intelligence aimed

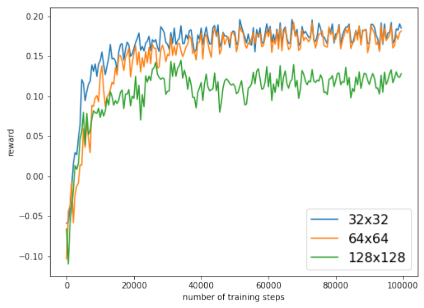

Figure 4: Tuning size of hidden layer

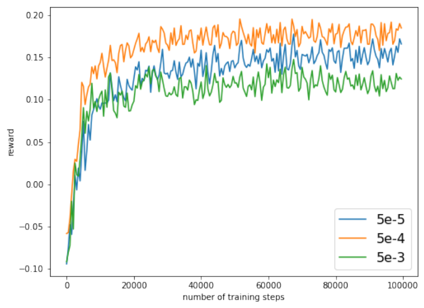

Figure 5: Tuning learning rate

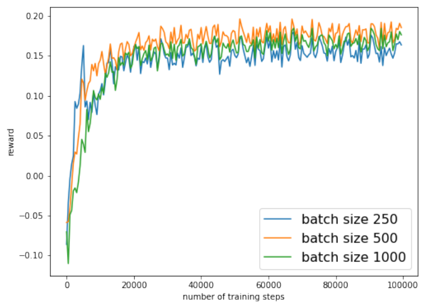

Figure 6: Tuning batch size

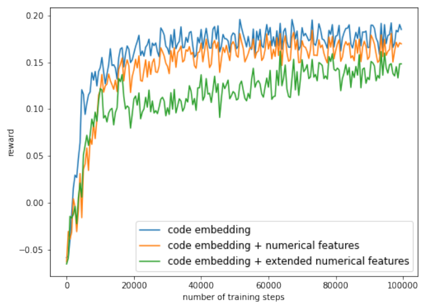

Figure 7: Reward of training when numerical features are added

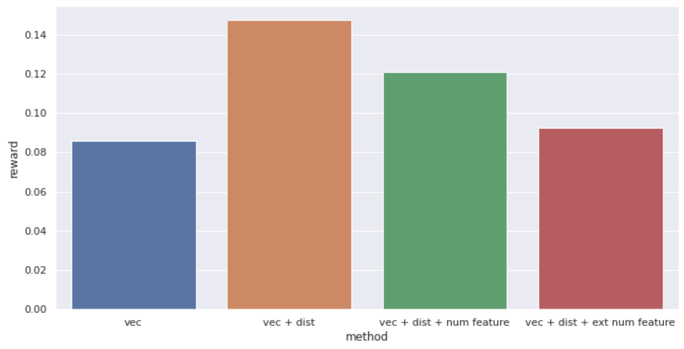

Figure 8: Reward of test

at detecting and predicting patterns. In fact, the use of machine learning in compiler optimization is a natural fit and has developed into an established research domain. Machine-learning compilation is becoming increasingly popular due to the automatic nature of machine learning and the need for new ways to bridge the software gap caused by the increasing potential performance of hardware [8].

Reinforcement learning is a specific type of machine learning that learns from trials and errors. There are a lot of factors and different optimization techniques in the field of compiler optimization. Reinforcement learning is a powerful tool to efficiently explore and improve different strategies by trying out as many optimization factors as possible. Ultimately, it will come up with an optimal policy that has been improved from past experience. There has been many studies and experiments on using RL to optimize compiler [7].

6 Conclusion

In this project, our goal is to use RL to generate vectorization and distribution optimization factors to reduce loop's execution time. We have applied Code2Vec framework and developed a LLVM pass to generate loop features of code. Then we have applied PPO, a RL method, to predict the optimal factors and inject vectorization and distribution pragmas into the source code. We have evaluated the performance with vectorization only and vectorization together with distribution, and noticed the performance was improved with distribution optimization. We tuned hyperparameters to find the best model. Furthermore, we experimented with different feature sets to evaluate the performance. Our result showed that adding some numerical static features generated from LLVM loop passes did not contribute to improving the optimization performance. Generating more features suitable for the loop optimization problem may be benefiting the optimization, which can be a future plan.

References

[1] ALON, U., ZILBERSTEIN, M., LEVY, O., AND YAHAV, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–29.

[2] GROSSER, T., GROESSLINGER, A., AND LENGAUER, C. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters 22*, 04 (2012), 1250010.

[3] HAJ-ALI, A., AHMED, N. K., WILLKE, T., SHAO, Y. S., ASANOVIC, K., AND STOICA, I. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (New York, NY, USA, 2020), CGO 2020, Association for Computing Machinery, p. 242–255.

[4] J. SCHULMAN, F. WOLSKI, P. D. A. R., AND KLIMOV, O. Proximal policy optimization algorithms. *arXiv preprint*, 1707.06347 (2017).

[5] LOMONT, C. Introduction to intel advanced vector extensions.

[6] MASTEN, M., TYURIN, E., MITROPOULOU, K., GARCIA, E., AND SAITO, H. Function/kernel vectorization via loop vectorizer. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)* (2018), pp. 39–48.

[7] TROFIN, M., QIAN, Y., BREVDO, E., LIN, Z., CHOROMANSKI, K., AND LI, D. Mlgo: a machine learning guided compiler optimizations framework, 2021.

[8] WANG, Z., AND O'BOYLE, M. Machine learning in compiler optimisation, 2018.