Learning to Vectorize Using Deep Reinforcement Learning

Ameer Haj-Ali[†,‡] **Nesreen K. Ahmed**[†] **Ted Willke**[†] **Sophia Shao**[‡] **Krste Asanovic**[‡] **Ion Stoica**[‡]
[†]Intel Labs, [‡]University of California, Berkeley
{ameerh, ysshao, krste, istoica}@berkeley.edu, {nesreen.k.ahmed, ted.willke}@intel.com

Abstract

We explore a novel approach for handling loop vectorization automatically and propose an end-to-end solution using deep reinforcement learning (RL). We conjecture that deep RL can capture different instructions, dependencies, and access patterns to enable learning a sophisticated model that can better predict the actual performance cost and determine the optimal vectorization factors when compared against the currently used fixed-cost models that rely on heuristics. We develop an end-to-end framework, from code to vectorization, that integrates deep RL in the LLVM compiler. Our proposed framework takes benchmark codes as input and extracts the loop codes. These loop codes are then fed to a loop embedding generator that learns an embedding for these loops. The learned embeddings are used as input to a Deep RL agent, which dynamically determines the vectorization factors for all the loops.We evaluate our approach against the currently used LLVM vectorizer and loop polyhedral optimization techniques. Our experiments show $1.29 \times -4.73\times$ performance speedup compared to baseline and only 3% worse than the brute-force search on a wide range of benchmarks.

1 Introduction

Modern computers typically have vector instructions that perform multiple basic operations simultaneously, such as Intel Advanced Vector Extensions (AVX) [8]. Converting a computer program from a scalar implementation, which processes a single pair of operands at a time to a vector implementation, which performs a single operation on multiple data (SIMD) items at once is called *vectorization*, and is critical to enhancing the performance of compute-intensive programs for modern computers.

Loops are among the most commonly vectorized parts of code. Loop vectorization is done by defining the vectorization factor (VF) and the interleaving factor (IF) [10]. VF determines how many instructions to pack together from different iterations of the loop. IF determines the stride of the memory accesses of the packed instructions. IF allows vectorization to be performed on non-consecutive addresses, which are generally referred to as non-unit stride accesses. In most C and C++ compilers it is possible to use intrinsic pragmas or compiler passes to manually vectorize loops by setting the VF and IF. However, manual vectorization is labor-intensive, error-prone, and results in code that is difficult to maintain and port. Alternatively, several solutions for automatic vectorization and loop optimization have been proposed. The current vectorizer used in LLVM and proposed improvements [15, 16], rely on linear and constant-cost models to predict the vectorization factors. Unfortunately, these cost models do not consider the computation graph and focus on estimating the cost of different instructions with predefined heuristics. Another commonly used approach is Polly [3]. Polly uses loop polyhedral analysis, which relies on an abstract mathematical representation, namely equations and matrices, to represent loops as polytopes. The polytope representation simplifies the implementation of loop optimizations, though to date the main optimizations in Polly are tiling and loop fusion to improve data locality. Machine learning is yet another recent approach that has been proposed for automatic vectorization [13]. While this approach improves the cost models implemented by existing compilers, they use hand-engineered heuristics to extract features from the assembly code, such as arithmetic intensity. Unfortunately, these features are typically not sufficient to fully capture the code functionality. We summarize the contributions of this paper as follows[1]:

[1]A full version of this paper is available on: https://arxiv.org/abs/1909.13639 and will appear on CGO 2020 [4].

NeurIPS 2019 Workshop on Machine Learning for Systems (NeurIPS 2019), Vancouver, Canada.

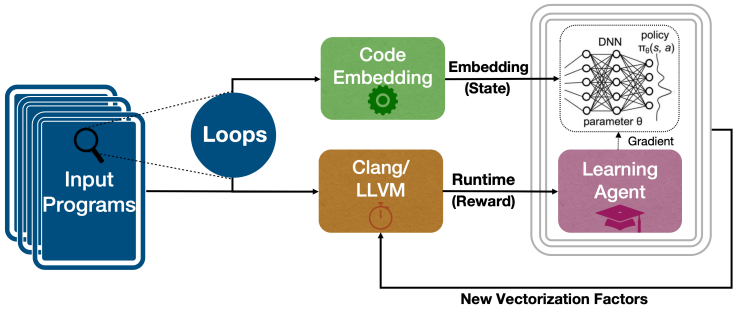

Figure 1: The proposed framework for automatic vectorization with deep RL. The programs are read to extract the loops. The loop texts are fed to the code embedding generator to generate an embedding. The embedding is fed to the RL agent. The RL agent learns a policy that maps this embedding to optimal vectorization factors by injecting compiler pragmas and compiling the programs with Clang/LLVM to gather the rewards: the execution time improvements.

```
int vec[512] __attribute__((aligned(16)));
__attribute__((noinline))
int example1 () {
    int sum = 0;
    for(int i = 0; i<512; i++){
        sum += vec[i]*vec[i];
    }
    return sum;
}
```

RL Agent's Action

```
int vec[512] __attribute__((aligned(16)));
__attribute__((noinline))
int example1 () {
    int sum = 0;
    #pragma clang loop vectorize_width(64)\\
    interleave_count(8)
    for(int i = 0; i<512; i++){
        sum += vec[i]*vec[i];
    }
    return sum;
}
```

Figure 2: An example of the automatically injected VF and IF pragmas by the RL agent.

- An end-to-end deep reinforcement learning (RL) [14] based auto-vectorization method. Unlike deep supervised learning approaches, deep RL does not require labels to train, can better handle large search spaces and co-optimize multiple objectives.
- An extensible framework that integrates learning code embedding with multiple machine learning methods to make vectorization predictions on loops. We explore using random search, nearest-neighbor search (NNS) [12], decision trees [11], and contextual bandits based on deep RL.
- An evaluation against the currently used cost model, as well as Polly. Our results show $1.29\times$ $-4.73\times$ average performance speedup and only 3% worse than the brute-force solution.

2 Proposed Automatic Vectorization Framework

The proposed framework for automatic vectorization with deep RL and its components are illustrated in Figure 1. The directory of code files is fed to the framework as text code. This code is fed to an automatic loop extractor. The extractor finds and outputs all the loops and their contexts in all the source codes. These outputs are fed to a code embedding generator to learn and generate an embedding. The latter is fed to the deep RL agent to predict the vectorization factors. The agent automatically injects vectorization pragmas as shown in Figure 2. The agent then compiles the program with clang/LLVM to gather the execution time improvements, which are used as rewards to the RL agent. Once the model is trained it can be plugged in as-is for inference without further retraining. Note that our framework cannot introduce new errors in the compiled code. Our framework injects pragmas only. These pragmas are used as hints to make vectorization decisions on the loops. However, sometimes the compiler can decide not to consider these pragmas if it is not feasible. For example, predicates and memory dependency can hinder reaching high VF and IF. In that case, if the agent accidentally injected bad pragmas, the compiler will ignore it.

It is also possible to vectorize from the command line by giving the passes *-force-vector-width=VF* and *-force-vector-interleave=IF*. However, we do not use this option as it restricts us to use a single VF and IF pair for the entire code, which is far from being optimal. Furthermore, the pragma is injected for the most inner loop in the case of nested loops. Next, we discuss the details of each component in the proposed framework.

Code Embedding. The ultimate goal of the code embedding generator is to learn a function that maps the input loop codes to a point in a latent multidimensional space where similar loop codes are

mapped to points close to each other in the latent multidimensional space. This can allow the RL agent to make similar vectorization decisions on similar codes using the learned embedding. There are multiple ways to generate/learn an embedding for the input code. One example is to use Polly's mathematical representation of loops as an embedding. We see this as a potential future direction.

In this work we use code2vec [1]. Code2vec is a neural network model that relies on natural language processing [2] and attention [17] for representing snippets of code as continuous distributed vectors. Code2vec represents a code snippet as a single fixed-length code vector, which can be used to predict semantic properties of the snippet. This vector is composed of 340 features that embed the program code based on the mapping the code2vec neural network learned. This vector captures many characteristics of the code, such as semantic similarities, combinations, and analogies. Code is first decomposed to a collection of paths in its abstract syntax tree. Then, the network simultaneously learns the atomic representation of each path while learning how to aggregate a set of them.

Dataset Description. We first trained our model with long running benchmarks that include code that is not restricted to loops only. It took a long time to train since for every pragma injected for a loop the whole program has to be recompiled and executed. This can work with enough resources (mainly many CPU cores to allow running many programs in parallel). However, to speed up the training, and make it more efficient, we built a dataset that includes loops only. We built generators that generate more than 10,000 synthetic loop examples automatically from the LLVM vectorization test-suite[2]. For example, some new tests are made by changing the names of the parameters, which was crucial for reducing noise in the code embedding generator as often the names of the parameters might bias the embedding. Another examples included the stride, the number of iterations, the functionality, the instructions, and the number of nested loops.

The RL Environment Definition. To learn a good policy, it is necessary to appropriately define actions, rewards and states. We define the agent's reward as follows:

$$reward = (t_{baseline} - t_{RL})/t_{baseline}, \tag{1}$$

where $t_{baseline}$ is the execution time when compiled with the currently implemented baseline cost model in LLVM and t_{RL} is the execution time when compiled with the injected pragmas by the RL agent. We normalize the execution time by $t_{baseline}$ so that our reward metric is robust to the variations in the programs' execution times. We also use $t_{baseline}$ as a bias in our reward so that a positive reward means the current configuration improves over the baseline. This also reduces the variance in the learned policy.

An action picks the VF and the IF, respectively, from the following values:

$$\begin{aligned} VF &\in [2^0, 2^1, 2^2, ..., \text{MAX\_VF}], \\ IF &\in [2^0, 2^1, 2^2, ..., \text{MAX\_IF}], \end{aligned} \tag{2}$$

where MAX_VF and MAX_IF are respectively the maximum VF and IF supported by the underlying architecture. Note that the actions for VF and IF can be defined to have values that are not powers of two. Here they were defined as powers of two only because this is what LLVM currently supports. Initially, we trained two agents, one that predicts VF and the other predicts IF independently. However, from our experiment combining these two agents into one agent with a single neural network that predicts the VF and IF simultaneously performed better. This also aligns with the fact that IF and VF are directly correlated, and in the LLVM compiler code they are defined as a function of each other.

The states of the RL agent were defined as the vector output embedding from the code embedding generator. For the inputs of the code embedding generator, we experimented with different snippets of the loop bodies and observed that for nested loops, feeding the loop body of the outermost loop, which also includes the bodies of the inner loops, performed better than feeding the body of the most inner loop only. This is mainly because the entire loop nest better captures the functionally of the code, and reveals the access patterns and strides.

During training, some of the programs took a long time to compile, mainly when the agent was trying to vectorize more than plausible. Long compilation time with limited resources can slow down the training. To overcome this, we limited the compilation time to ten times the time it takes to compile a program with the baseline cost model. If the program took longer than that to compile, we gave a penalty reward of -9 (equivalent to assuming it takes ten times the execution time of the baseline) so

[2]The test suite is available on: https://github.com/llvm/llvm-test-suite/tree/master/SingleSource/UnitTests/Vectorizer.

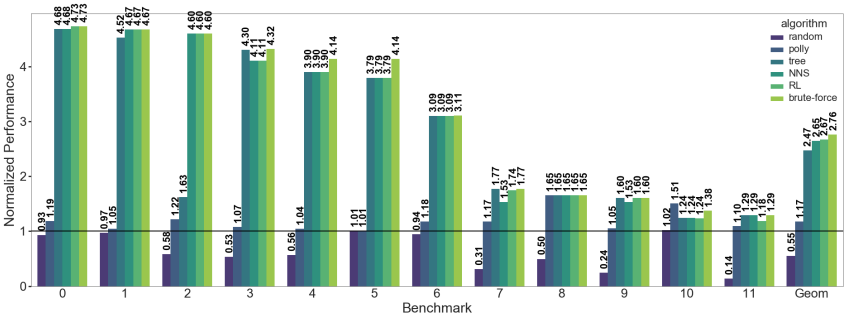

Figure 3: The performance of the proposed vectorizer that can be configured to use NNS, random search, decision trees, and RL compared to brute-force search, Polly and the baseline cost model. The performance is normalized to the baseline.

that the agent will learn not to overestimate the vectorization and avoid it. From our experiments on the programs that took a relatively long time to compile, eventually after waiting the necessary time for them to compile, the achieved performance was not better than that of all the other possible vectorization configurations. In some contexts, users might care about compile-time when evaluating the performance of programs. Our reward definition can incorporate that too so that the agent can simultaneously optimize for more than one objective. For example, one can allow a long compilation time but penalize for it. The reward can also be defined as a combination of the compilation time, execution time, generated assembly code size, *etc*.

3 Experiments and Conclusion

The proposed framework is evaluated on 16 GB 2133 MHz LPDDR3 memory and 2.7 GHz (up to 4.5 GHz) Intel 4-Core i7-8559U [5], which supports AVX. For code2vec we use the open-source code and modify it to work with our RL agent implementation. To run our RL algorithms we use RLlib [6] and Tune [7], open-source libraries for RL that offer, high scalability, hyperparameter tuning and a unified API for a variety of applications. RLlib and Tune are built on top of Ray [9], a high-performance distributed execution framework targeted at large-scale machine learning and RL applications. We first train the framework with the RL agent and code2vec until convergence. Then we also run a brute-force search on the dataset to find the best vectorization factors and use them as labels for NNS and the decision tree. Since the brute-force search requires a long time to run, we limit our training set to 5,000 samples and use this set for the rest of our evaluation. From our training set we keep out 20% of the samples for testing. To report performance we take twelve completely different benchmarks from the test set. These benchmarks combine completely different benchmarks from the LLVM test-suite. These benchmarks include loops with different functionality and access patterns. For example, predicates, memory accesses with different strides, bitwise operations, unknown loop bounds, if statements, unknown misalignment, multidimensional arrays, summation reduction, type conversions, different data types, *etc*. We compare the performance of our framework versus Polly and the baseline cost model.

The performance on different benchmarks for the baseline, random search, Polly, decision tree, RL and brute-force search are shown in Figure 3. RL outperformed the baseline by $2.67\times$ on average and achieved performance only 3% worse than that of the brute-force search. The performance differed between the different benchmarks based on how much vectorization the program can absorb. NNS and decision trees also performed well, achieving respectively $2.65\times$ and $2.47\times$ better than the baseline. This shows that the embedding learned by the code embedding generator during the end to end training with the RL agent is good so that other learning methods that cannot be trained end to end can use this embedding and perform well. Random search performed much worse than the baseline. This shows that the framework learned a structure in the observations that manifested in the vectorization decisions it made. Polly outperformed the baseline by 17% but performed 56% worse than the proposed RL solution. For benchmark #10, Polly interestingly outperforms the brute-force search. This is because Polly performs loop transformations that optimize beyond vectorization. This also shows the potential for achieving better performance improvement when combining Polly and deep RL. We plan to explore this option in future work.

Conclusion. In this work, we proposed and developed an end-to-end vectorization framework that automatically detects loops, learns their structures and applies deep RL to inject vectorization pragmas to the compiler. Our results demonstrated an average performance improvement $1.29\times - 4.73\times$

compared to the baseline cost model implemented in LLVM and on average only 3% worse than the brute-force solution. Looking forward, we foresee a potential opportunity for automatic end-to-end code tuning and optimization with machine learning techniques, such as deep RL.

References

[1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019.

[2] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[3] T. Grosser, A. Groesslinger, and C. Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[4] A. Haj-Ali, N. Ahmed, T. Willke, S. Shao, K. Asanovic, and I. Stoica. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 2020 International Symposium on Code Generation and Optimization*, CGO 2020. ACM, 2020.

[5] Intel Inc. Intel Core i7-8559U Processor Specification, 2018.

[6] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica. Ray rllib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 2017.

[7] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[8] C. Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, pages 1–21, 2011.

[9] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging ai applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[10] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. *ACM SIGPLAN Notices*, 41(6):132–143, 2006.

[11] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[12] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM sigmod record*, volume 24, pages 71–79. ACM, 1995.

[13] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):50, 2012.

[14] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[15] X. Tian, H. Saito, E. Su, A. Gaba, M. Masten, E. Garcia, and A. 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. IEEE, 2016.

[16] K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337. IEEE, 2009.

[17] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.