NeuroSLP: A Reinforcement Learning Heuristic for Superword-Level Parallelism

Kyle Astroth, Samuel Gonzalez, Carson Hoffman, Xiangyu Qin

***Abstract*—Superword-level parallelism (SLP) is a highly general technique used to take advantage of hardware support for vectorization across arbitrary sequences of instructions, rather than only in select scenarios such as loops. Existing works have utilized machine learning to improve loop vectorization heuristics in the past, however these works are limited and minimal effort has been made to leverage AI in improving SLP vectorization. In this paper we extend the work done by NeuroVectorizer [1] and introduce NeuroSLP, an open-source deep reinforcement learning framework capable of outperforming clang-14's default parameters for SLP vectorization. When evaluated on range of 1,000 test programs, NeuroSLP is capable of improving runtime performance by over 6.5% on average and up to 54% in select applications.**

***Index Terms*—Automated Vectorization, Superword Level Parallelism, Deep Reinforcement Learning, Compiler Optimization**

This project is open source. A complete copy of the source code is available at https://github.com/samrg123/NeuroSLP

I. Introduction

While loop vectorization presents a clear definition of parameters and what may be vectorizable, the same does not exist for superword-level parallelism (SLP). Because it attempts to deduce vectorizable sequences of instructions on a basic block level, there is no simple manipulation to the source code which can be applied to select vectorization parameters (such as pragmas for loops). Consequently, parameters for SLP vectorization are currently selected at a program-wide level, rather than a more granular per-loop or per-block level, meaning that the effects of parameter choices must be considered across an entire program. Although this may limit the effectiveness of any parameter selection scheme, certain programs can still benefit from individualized parameter selection as shown in figure 1.

A. Previous Work

NeuroVectorizer [1] is an existing open-source framework which applies the technique of reinforcement learning (RL) to the selection of parameters for loop vectorization. It takes source code as input, translates it to a representation more compatible with machine learning called an embedding via another machine learning model named **code2vec** [2], then observes the runtimes of the program with various parameter choices. The embedding is a vector of numeric values attempting to synthesize information relevant to the program. Crucially, the embeddings generator is trained to produce similar vectors for source code which feature similar attributes with respect to vectorization. We began to investigate the use of code2vec in our own work under the hypothesis that the principle of transfer learning may be applicable—that although NeroVectorizer was trained under the goal of improving loop vectorization, the knowledge it gained in doing so may allow the embeddings it produces to be useful for the purposes of improving SLP vectorization.

II. Experimental Setup

A. Data set

NeuroVectorizer has provided a data set with 10,000 synthetic loops on which they trained their RL network. We will be making use of the same data set, but first unroll the loops in order to obtain a data set that is more representative of when SLP vectorization would be utilized.

B. Pre-processing

The proposed framework for automatic vectorization with deep RL and its components are illustrated in figure 2.
To best utilize the data set for training the deep RL model on SLP vectorization, we first use clang pragmas to unroll the loops 512 times. Once the data is in the desired form, we pass it into code2vec as input to obtain a code embedding as output, that is later fed into the RL model to learn features from.

As discussed previously, although code2vec was trained on the 10,000 synthetics loop, prior to being unrolled, we believe that we could apply transfer learning to reuse the same neural network without retraining the whole model. This is because regardless of whether the loops are unrolled or not, the basic key component of code would not be affected. That is, for example, an *if* statement would still maintain its meaning regardless the presence of loops.

C. Parameter Selection

Clang offers 12 compiler argument parameters related to SLP vectorization. In order to prevent over complicating the model, we hand picked two of the SLP parameters to train the neural network on. We made this decision by compiling a few sample input files using a range of values for each compiler argument. We observed that parameters like slp-upper-bound, which has a very high default value of 512 showed little to no change in the binary file compiled so we stepped away from any parameters that had a large range of possible values. Furthermore, we believed that many of the boolean flag arguments did not strongly correlate with the state of the code. This is because arguments like slp-vectorize-hor-store are simply a flag to enable more aggressive SLP vectorization that is off by default which would give NeuroSLP an unfair advantage.

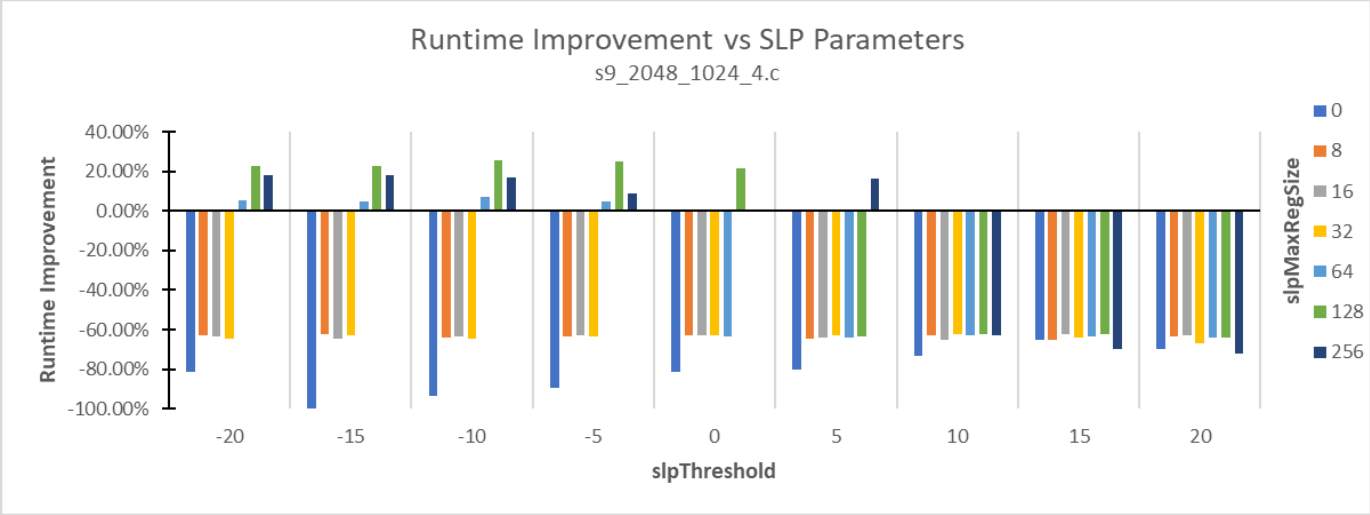

Fig. 1: A distribution of the s9_20048_1024.c test loop runtime improvement compiled with all combinations of slp-threshold and slp-max-reg-size. Notice how LLVM's default 256-bit slp-max-reg-size parameter is sub-optimal and that decreasing the register size to 128 yields 25% improvement despite halving the amount of data operated on in each instruction.

After these eliminations, the following four parameters remained: slp-threshold, slp-max-reg-size, slp-max-vf, and slp-min-tree-size. Out of the four remaining parameters, we determined that slp-threshold and slp-max-reg-size offered the most improvement. Figure 1 displays the result of a brute force search across different compiler argument values for each of the chosen parameters. From here it can be observed that both parameters contribute to improvements, as we can see fluctuation in the runtime even if one parameter is fixed.

D. Optimization

We estimated that each time the deep RL model is trained, it would take over 190 hours for the training to complete. Because we wanted to experiment with different SLP and RL hyperparameters, using the existing architecture would be infeasible and required us to optimize NeuroVectorizer training.

When investigating where the bottleneck was, we realized that the majority of the computation time corresponded to compiling of the code with the different compiler arguments and running the binary files to obtain the runtime data. This unfortunately means that using a GPU would not improve the training time unlike many of the other machine learning models since the GPU is not suited for code compilation. Instead, we incorporated two optimizations to reduce the number of times we would have to compile and run the code. The first optimization is caching the runtimes and binary file of each compiler argument set across multiple runs of the deep RL network. There were two reasons for this decision. First, the binary file is static: it does not change regardless of what condition it was compiled in as long as the compiler arguments are consistent. Second, this allows debugging and evaluation of the results easier as we are able to dive deeper into the *why* and *how* certain compiler arguments have performed better than others.

For caching the runtime of the source code compiled under each compiler argument combination we had two options, each with their trade-offs. Similar to caching the binary files, we could run each program once and store its runtime in a separate file that would have to be manually deleted by the user. Alternativly we could keep the runtime cached throughout the training iterations, but delete the cache between invocations of deep RL training. The first option provides consistency across different invocations of the training and can be helpful in evaluating and reproducing results. The second option provides a more accurate runtime based on the current state of the processor. For example, we examined that the runtime of the code is greatly affected by how much of the CPU is currently used. Given that trainings could be invoked on completely independent occasions, we determined that the second option, although sacrificing some optimization in computation time, would yield more accurate results, and thus was more desirable.

Because we are now only running the binary file on one occasion, any noise when running the binary file could have tremendous effect on how the model learns, as we will be reusing the same runtime value for each combination of compiler arguments. To overcome this, when obtaining the run time of the binary file, we executed the code three times, and took the average amongst them.

When evaluating our results, we observed that many files compiled to identical binaries as LLVM's default parameters, but had runtimes that fluctated $0 \pm 2\%$. To reduce noise or reporting slight improvement or loss when there is no change at all, we further extended our optimization to check whether the compiled binary file was identical to that of LLVM's default parameters and returned the same runtime if they were. This way, we successfully reduced noise in our results, and were

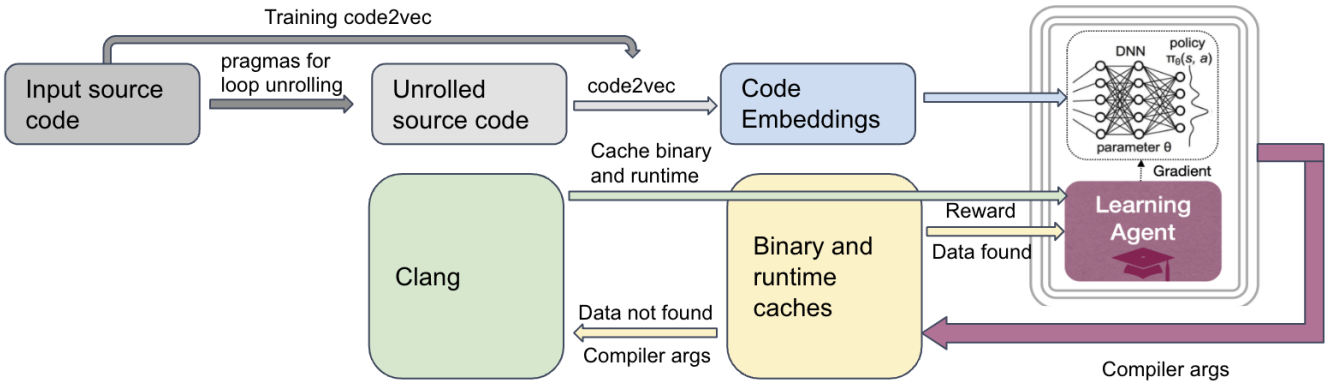

Fig. 2: The proposed deep RL framework for automatic SLP vectorization. The input programs with simple loops are first unrolled using pragrams. The unrolled source code are then fed into the code embedding generator to generate an embedding, which is then fed to the RL agent. The RL agent learns a policy that maps this embedding to optimal max-slp-reg-size and slp-threshold parameters. Given these parameters, we first check for whether we have the binary file and run-time cached. If there is a cache hit, we return the reward into the learning agent. Otherwise, we compile the source code with the provided parameters, run the compiled binary, cache the binary and run-time data, and return reward into the learning agent.

able to speed up computation by preventing multiple runs of identical binary files.

III. Evaluation

To evaluate NeuroSLP the latest stable version of clang-14 as of testing was used to compile programs. All programs were compiled with $commonArgs$, as shown in equation 1, to disable loop vectorization and ensure that only SLP vectorization was being evaluated.

$$\begin{aligned} commonArgs = [\quad & -O3 \\ & -march = native \\ & -fno\text{–}unroll\text{–}loops \\ & -fno\text{–}vectorize \\ & -fslp\text{–}vectorize \quad] \end{aligned} \tag{1}$$

A. Training

NeuroSLP was trained on 1,000 random loops taken from the NeuroVectorizer dataset. Training was performed with a batch size of 500 for a total of 200 epochs and 100,000 episodes using the optimal learning rate of $1e-4$ discovered in the NeuroVectorizer paper.

Likewise, program runtime improvement was was chosen as a reward for NeuroSLP's parameter selection as shown in equation 2,

$$reward = \frac{tBaseline - tNeuroSLP}{tBaseline} \tag{2}$$

where $tBaseline$ is the average runtime of the program using LLVM's default SLP parameters, and $tNeuroSLP$ is the average runtime of the program compiled with NeuroSLP's selections for slp-threshold and slp-max-reg-size.

To further minimize noise in program runtime, NeuroSLP was trained on a single thread on an computer left under idle load. This dramatically increased training time as no parallelization was performed, and required 56 hours to complete on an Intel Core i7-8750H processor. As seen in figure 3, the neural network quickly convergences to match the performance of LLVM's default parameters in 30 epochs, but requires significantly more epochs before any noticeable improvements are seen. By the end of 200 epochs, every loop compiled with NeuroSLP's parameters performed just as well as or outperformed LLVM's default parameters. Despite this, we suspect there is still headroom for NeuroSLP to perform even better given a larger number of training epochs, but were unable to test this claim given our limited time.

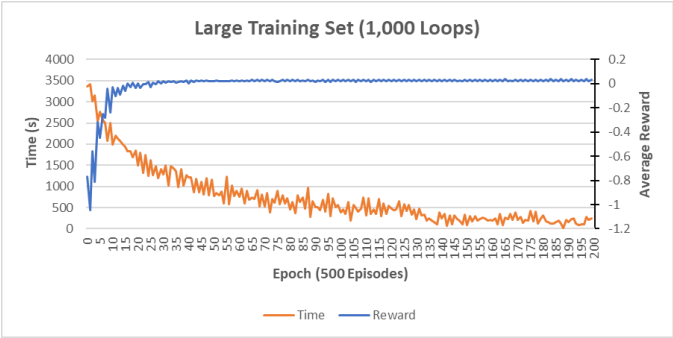

Fig. 3: The training time and reward graphed over each epoch while training NeuroSLP on 1,000 loops over a period of 56 hours. Training time decreases exponentially with each epoch due to caching. Without caching, training NeuroSLP would have taken an estimated 194 hours to complete.

B. Testing

To test NeuroSLP another random 1,000 loops not used for training were taken from the NeuroVectorizer dataset and

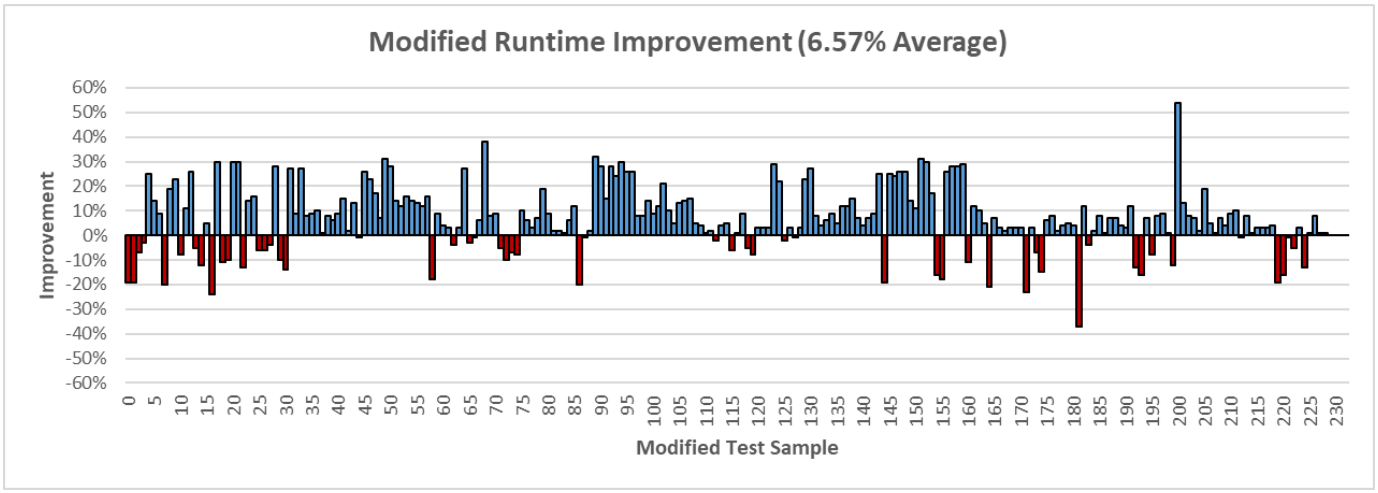

Fig. 4: The distribution of average runtime improvements of 230 modified binaries output from NeuroSLP during our evaluation. On average we see a 6.57% improvement when compared to LLVM's default SLP parameters with select applications executing up to 35% faster. Its important to note that the increases in runtime depicted in red largely correspond to NeuroSLP disabling vectorization altogether and can be mitigated by limiting slp-max-reg-size to be larger than 64 bits.

compiled with the selected SLP parameters outputted from the neural network. The runtime of these loops were then compared to the runtime of the loops compiled with LLVM's default SLP parameters and the percent improvement was calculated using equation 3.

$$\%improvement = 100 \cdot \frac{tBaseline - tNeuroSLP}{tNeuroSLP} \quad (3)$$

From our testing, NeuroSLP achieved an average improvement of 1.53% across all 1,000 loops. However, 77% of the time NeuroSLP resulted in the same binary output as the default LLVM parameters. To better understand how NeuroSLP performs, we filtered out the unchanged binaries and only compared the remaining modified programs in figure 4. This yielded a much better 6.57% average improvement with the median program showing a 5.66% reduction in runtime.

We also observed that select programs significantly benefited from NeuroSLP and outperformed LLVM's default parameters by up to 35% when choosing optimal SLP parameters. Despite this, NeuroSLP negatively impacted runtime performance on 24% of modified binaries when choosing SLP parameter combinations that effectively turn off vectorization. This leads us to believe that further improvement could be obtained through more training, or by restricting NeuroSLP's output for slp-max-reg-size to be larger than 64-bits, thus preventing it from disabling vectorization.

IV. Limitations

NeuroSLP produced meaningful results, but we see several areas for improvement due to current time and resource limitations. One of the most notable and expected limitations our project faced was an extremely lengthy training time, even with a relatively small number of epochs. Because long training times are expected for many machine learning models, it is common to try to parallelize or distribute training with multiple CPU cores or a GPU. However, our model's reinforcement learning agent relies on program runtime so running multiple programs concurrently wouldn't provide accurate CPU metrics. So we were limited to one CPU core and couldn't take advantage of a GPU. Similarly, while NeuroVectorizer was trained on a dataset of 10,000 samples, due to single core training time constraints, NeuroSLP was trained on only 1,000 samples. In addition the training set, which was taken from LLVM's test suite, consisted entirely of loops. Such a dataset is not representative of most SLP workloads, and many programs that could most directly benefit from SLP vectorization are not included in it. Further, we opted not to retrain code2vec, and instead take a transfer learning approach and use the pre-trained code2vec model provided by NeuroVectorizer. We predict NeuroSLP's optimizations could be improved by first compiling a more diverse dataset of samples that would benefit from SLP optimization and retraining code2vec and the reinforcement learning model on the larger dataset.

As mentioned above, another main limitation was retrieving accurate CPU metrics that hadn't been polluted by background processes when attempting to measure the runtime of a single program. We took several measures to reduce noise, such as caching binary files and decreasing the number of workers set by **Ray** [3] from one to zero (reducing the number of threads from two to one), but the noise of the metrics will inevitably increase with training time. Performance of a model and its evaluation metrics will degrade over time, and our model could potentially benefit from a noisy metric threshold that could flag degrading runtime accuracy.

Finally, the scope of our implementation was limited by LLVM only allowing for control of SLP parameters for an entire program, and thus preventing us from implementing SLP at the basic block level. We predict that being able to dictate SLP parameters individually to basic blocks would also further

improve NeuroSLP's optimizations.

V. Future Work

As discussed in section IV, the most obvious next steps that should be taken to improve NeuroSLP's performance are to gather a larger and more SLP relevant dataset, and retrain code2vec and the reinforcement learning model accordingly. Additionally, implementing further noise reduction measures, such as a threshold or a neural network for noise suppression, could be implemented for cleaner CPU metrics. Writing an LLVM compiler pass that allows us to pass SLP parameters to individual basic blocks would further increase performance.

Some additional ideas that our team considered were replacing the code2vec embedding with an LLVM abstract syntax tree representation, such as the one provided by **Polly** [4]. A tree representation would likely reduce overhead and possibly be a more accurate program representation, but could also potentially lack some of the learned insights provided by code2vec, such as such as semantic similarities, combinations, and analogies.

We also considered ideas for future improvements proposed by the authors of NeuroVectorizer. First, they proposed replacing code2vec with loop polyhedral analysis (Polly), which would similarly be less computationally expensive. Second, they considered combining their RL model with Polly to further boost the performance of the learning agent. They also considered employing the model at the intermediate representation level, which could potentially better reflect vectorization and stronger predictions. On a broader level, future work in the area could be explored by implementing different machine learning models, such as different neural network architectures.

VI. Conclusion

We successfully extended the end-to-end NeuroVectorizer framework to support superword level parallelism, and achieved performance improvements of up to 54%. We found the NeuroVectorizer model to make a more broadly significant contribution to decreasing runtime, but for niche applications and functions NeuroSLP makes considerable improvements. We propose widening the provided dataset, implementing NeuroSLP in tandem with NeuroVectorizer to take advantage of both optimizations, and possibly incorporating further optimizations, such as Polly, in the reinforcement learning model.

Appendix

Acknowledgments: Special thanks to Professor Mahlke, Sanjay Singapuram, and Yunjie Pan for the great learning experience!

References

[1] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. ACM, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3368826.3377928

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290353

[3] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging AI applications," *CoRR*, vol. abs/1712.05889, 2017. [Online]. Available: http://arxiv.org/abs/1712.05889

[4] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly — performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.