OpLUn: Optimal Loop Unrolling using Neural Networks

Rishi Baronia, Akanksha Girdhar, Aarya Kulshrestha, Aidan Szuch, Omkar Vodela
University of Michigan
Ann Arbor, MI
{rbaronia, agirdhar, akulshre, amszuch, omkarv}@umich.edu

***Abstract*—Loop unrolling is a critical compiler optimization technique that can significantly improve runtime performance by reducing loop control overhead. However, determining the optimal unroll factor remains a challenging problem due to the complex interactions between loop structures, data dependencies, and hardware characteristics. This paper introduces OpLUn, a neural network-based approach to predicting optimal loop unrolling factors using machine learning.**

The research develops a Multilayer Perceptron (MLP) classifier trained on a comprehensive dataset of 591 C source files, utilizing 22 carefully selected loop-specific features. Through extensive hyperparameter tuning and training, the model achieves a training accuracy of 86.32% and a test accuracy of 65.28%. The approach was evaluated on the PolyBench benchmark suite, demonstrating the potential of data-driven methods to improve compiler optimization strategies.

While the results highlight challenges such as dataset imbalance and runtime measurement noise, the study provides promising evidence that machine learning can enhance traditional heuristic-based compiler optimization techniques. The research points to future directions for developing more robust and comprehensive models for loop unrolling optimization.

Changes since presentation on Dec. 4:
- Expansion of dataset
- Reintegration of some loop features
- Improvements to runtime evaluation
- Evaluation on PolyBench

I. Introduction

Loop unrolling is a well-established optimization technique used by compilers to improve runtime performance through reducing the overhead associated with loop control. Loop unrolling works through replicating the loop body multiple times and in turn decreasing the number of iterations. The number of replications is known as the unroll factor, and optimal unroll factors can increase instruction-level parallelism to allow more concurrent instruction execution, optimize memory access patterns, and reduce the need for branching.

However, the efficacy of loop unrolling is highly dependent on contextual factors such as the loop structure, the presence of data dependencies, and even the underlying hardware. Different cases of loop unrolling may yield increased code size, register pressure and lead to register spills, negating any optimizations made.

Given the potential loop unrolling has, obtaining the optimal unroll factor is non-trivial in compiler optimization. Generally, unroll factors have been decided on the basis of different heuristics and thresholds, which is likely not sufficient in a diverse set of cases. As a result, compilers often select suboptimal unroll factors which may either have adverse effects or fail to capitalize on potential improvements. These challenges motivate the need for a different approach to select the unroll factor.

This paper explores a neural network-based solution, OpLUn, for predicting (Op)timal (L)oop (Un)rolling factors, by supplying the compiler with a data-driven, machine learning model to ease and more importantly improve the decision-making process. The model leverages a Multilayer Perceptron (MLP) classifier that learns from an expansive set of loop features to enable more precise predictions. Through comprehensive training, tuning, as well as validation from benchmarks such as PolyBench, the solution delivers substantial improvements to both single and multiple-loop test cases.

By incorporating OpLUn into more real-world applications in the compiler space, we aim to bridge the gap between heuristic and ML-driven optimization strategies to result in more effective and efficient compiler passes.

II. Related Work

The problem of optimizing loop transformations, such as loop unrolling, has been explored extensively in compiler optimization literature. Traditional approaches often rely on fixed heuristics or cost models that make decisions based on predefined rules. For example, the Open Research Compiler employs manual heuristics to determine unroll factors, which, while effective to some extent, can fail to capture complex interactions between loop structures and hardware features [1]. Similarly, Polly, an LLVM-based optimization framework, uses mathematical polyhedral analysis to model loops and optimize for memory access patterns, primarily targeting tiling and fusion transformations [2].

The advent of machine learning introduced novel ways to model such compiler decisions. Techniques such as support vector machines (SVMs) and supervised neural networks have been utilized to predict optimal unroll factors based on loop features [1]. Stephenson et al. demonstrated that SVMs could predict the unroll factor with 65% accuracy on a dataset of loops, outperforming traditional heuristic-based methods by achieving up to a 5% performance improvement for certain benchmarks [1].

More recently, reinforcement learning (RL) has been applied to loop optimization, allowing models to dynamically explore decisions rather than relying on pre-labeled datasets. Haj-Ali et al.'s NeuroVectorizer framework used deep reinforcement learning for automatic loop vectorization, achieving significant speedups (up to 4.73×) over heuristic-based LLVM vectorizers [2]. This work highlights the ability of RL to co-optimize multiple objectives, such as execution time and code size, while adapting to diverse loop structures [2].

These advancements underline the transition from heuristic-based to data-driven approaches in compiler optimization, with machine learning techniques offering more adaptable and scalable solutions for challenges like loop unrolling. By building upon these foundations, our work further explores how neural networks can enhance compiler decision-making by predicting the optimal unrolling factors, leveraging large-scale loop datasets and real-world benchmarks.

III. Method

A. Dataset

To develop a robust neural network capable of predicting the optimal unrolling factor for loops, a self-curated data set was meticulously constructed using 591 C source files. The decision to use C programs as the foundation for this data set was informed by prior research [1]–[5], where C/C++ programs were commonly used to construct data sets and benchmarks. Many other languages like C++, Java, Python, etc. are acquired directly or indirectly from C language itself [6]. Furthermore, C programs facilitate the creation of efficient compiler passes and executables, allowing an efficient data collection process for our experiments. The loops within these files were designed to exhibit diverse characteristics, including variations in loop structures, nesting levels, memory and data dependencies, and control flow. This diversity was critical to constructing a comprehensive dataset that mirrors the complexity of real-world programming scenarios, ultimately supporting the development of a generalizable predictive model for loop unrolling.

The process of dataset construction began with parsing each C file to identify and extract all unrollable loops. For every identified loop, a feature vector of 22 loop-specific features was extracted. The exact set of features can be seen in the Appendix at Features 1. These features were selected based on insights from prior research [1] to capture the critical aspects of loop structure, execution characteristics, and memory dependencies. The feature extraction process used custom-built LLVM passes and static analysis methodologies, inspired by techniques from other studies [1]–[6]. Certain features, such as critical path latency and cycle length estimates, were calculated using LLVM's Target Transform Info class, which abstracts hardware-specific details and exposes them to optimization passes based off the target architecture that LLVM is compiling for. We did not have access to hardware accelerators to get accurate runtimes to derive realistic feature values as other research did [5]. By combining static analysis with LLVM-based feature extraction, a comprehensive set of 22 features was captured for each loop in the dataset, allowing a detailed representation of the loop characteristics.

Once the feature vectors were generated, each loop was evaluated using multiple unroll factors from a set of predefined unroll factors for which we wanted to test. Each loop was executed five times for each unroll factor to account for natural variability in execution time. The average execution time for each unroll factor was calculated, and the unroll factor that produced the shortest average execution time was selected as the optimal label for that loop. This process ensured that the dataset contained high-quality ground-truth labels to support the training of an effective neural network classifier.

B. Model

1) Architecture: We chose to implement a classifier because, unlike regression tasks where the output is continuous, the unrolling factors for our scope are categorical values. Specifically, the unrolling factor can take one of five distinct integer values: 1, 2, 4, 6, or 8. Given that these values represent different levels of loop unrolling, and since the task is to predict a specific level of unrolling rather than a continuous value, classification was a natural fit.

For the neural network selection, we implemented a MLP classifier, which is a type of feedforward neural network, for its ability to learn complex nonlinear relationships between the input features and the discrete unrolling factors.

- Input Layer: The input layer consists of 22 neurons each representing a single loop feature, and the input layer passes these values to the first hidden layer.
- Hidden Layers: The MLP has two fully connected hidden layers with ReLU activation functions, allowing the model to capture complex patterns in the data.
- Output Layer: The output layer consists of 5 units, corresponding to the 5 distinct classes (1, 2, 4, 6, 8), with a softmax activation applied to produce class probabilities.

The Cross-Entropy Loss function was used to train the model, as it is well-suited for classification tasks with multiple classes. The output of the model is a probability distribution over the five classes, and the predicted class is the one with the highest probability.

2) Hyperparameter Tuning: We conducted a grid search to identify the best hyperparameters for the MLP model, including the number and size of hidden layers, learning rate, batch size, and the number of training epochs. The search was performed over the following parameter grid:

- Hidden Layer Sizes: [64, 32], [128, 64], [256, 128]
- Learning Rates: 1e-4, 1e-3, 1e-2
- Batch Sizes: 32, 64, 128
- Epochs: 100, 200

For each combination of hyperparameters, the model was trained, and the training accuracy was computed. The combination yielding the highest training accuracy was selected as the best set of hyperparameters. To better understand the relationship between hyperparameters and model performance, we used a parallel coordinate plot (Figure 1) to visualize the results of the grid search.

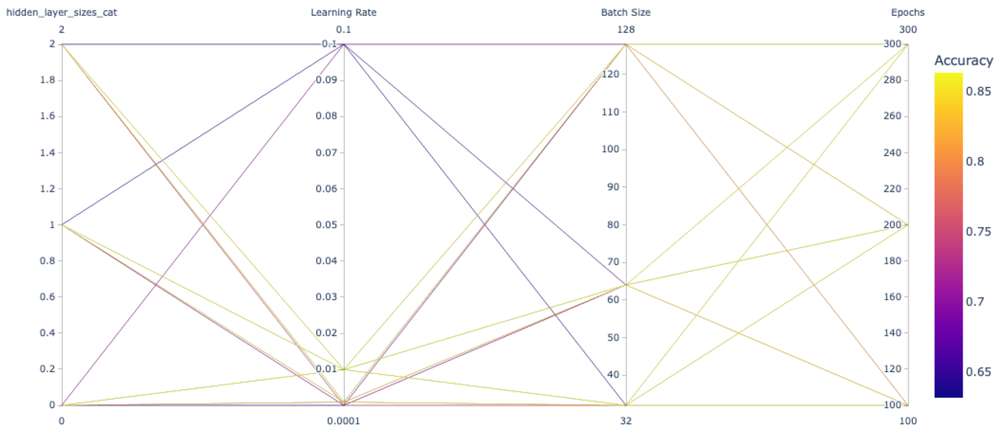

Fig. 1: Parallel coordinate plot to analyze the impact different hyperparameter combinations (hidden layer sizes, learning rate, batch size, and epochs) on model performance. Each line represents a unique set of hyperparameters, with the color indicating training accuracy

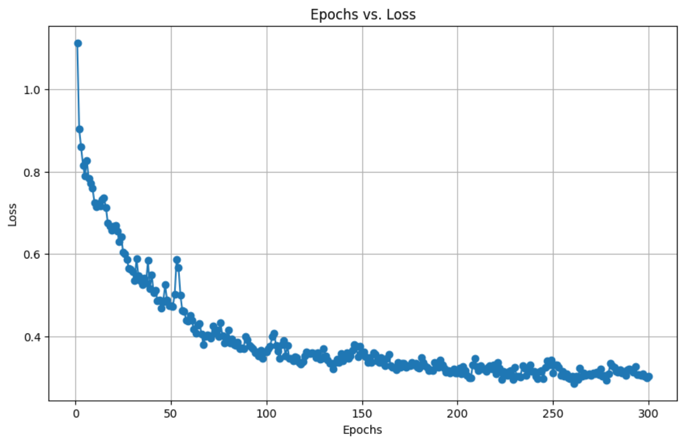

Fig. 2: Relationship between the number of epochs and the training loss. As the number of epochs increases, the model's loss gradually decreases, indicating successful convergence.

Components of the plot:

- X-axis: Each axis represents one of the hyperparameters
- Y-axis: Each line in the plot corresponds to a hyperparameter configuration, with the vertical positions of the lines reflecting the corresponding values for each hyperparameter.
- Color-Coding: The lines were color-coded based on the training accuracy to identify which combinations of hyperparameters led to the best performance.

We found the best training accuracy for the following hyperparameter combination:

- Hidden Layer Sizes: [128, 64]
- Learning Rate: 0.01
- Batch Size: 32
- Epochs: 300

3) Training: The model was trained using the Adam optimizer, which is known for its efficiency in training deep learning models. We used a batch size and learning rate from the best hyperparameters obtained from grid search, and trained the model for 300 epochs. During each epoch, the model's parameters were updated using backpropagation.

To track the model's learning progression, we looked at the loss over time. As shown in Figure 2, the loss decreased steadily with each epoch, which means the model was learning and improving. The final training loss was 0.2928, and the model achieved a training accuracy of 86.32%, suggesting that the model was well-optimized for the training data.

4) Model Deployment: Once we achieved 86.32% training accuracy, the model with the appropriate weights and the StandardScaler used for feature scaling was saved.

C. Evaluation

In order to evaluate our model, we began by looking at accuracy, precision, and recall for our model on the test

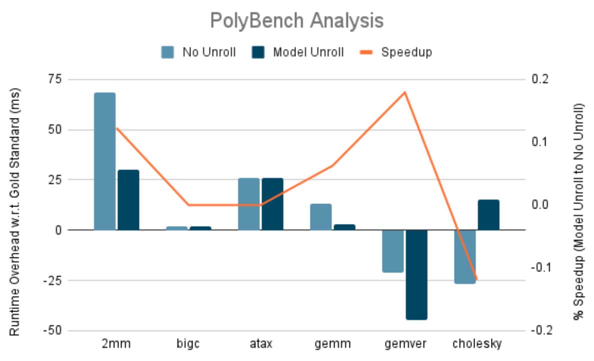

Fig. 3: Subset of runtime analysis of PolyBench. Left axis shows time in milliseconds the algorithm took compared to the gold standard with no unrolling or the model's predicted unrolling. Right axis shows speedup percentage from model compared to no unrolling.

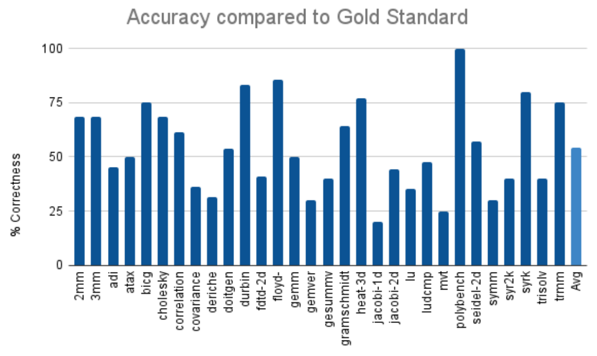

Fig. 4: Accuracy of model prediction compared to gold standard. Reflects whether or not model predicted the predetermined best loop unroll factor for each loop in benchmark.

data. We ultimately achieved a test accuracy of 65.28% for the model we selected, with precision of 69.49%, recall of 65.28%, and an F1-score of 66.12% (Table I). These numbers indicate that our model was slightly more inclined to prevent false positives than ensure true positives, but was overall fairly well balanced between precision and recall. While we would have liked to have higher values, failure to unroll enough or unrolling too much both reflect under-optimization, so having a balanced precision and recall is appropriate.

TABLE I: Test Metric Values

Test Metric	Value
Accuracy	0.6528
Precision	0.6949
Recall	0.6528
F1 Score	0.6612

To further assess the robustness of our feature set, we conducted an additional experiment inspired by prior research. Notably, a paper emphasized the importance of instruction count features in their work [3]. We developed another LLVM pass to extract a set of just instruction counts, analogous to our original dataset features. These extracted counts, detailed in the Appendix at Features 2, provide a granular view of loop characteristics at the instruction level.

To further expand beyond the dataset, we opted to also evaluate the model on the PolyBench suite [7]. For each of the algorithms provided, we began by establishing a gold standard for loop unrolling by running each loop with each of the possible unroll factors (1, 2, 4, 6, 8) and selecting the unroll factor that yielded the lowest average runtime. We then extracted the features of each loop, passed them through our model to get a predicted list of unroll factors, and determined the accuracy. We also opted to analyze runtime performance between no unrolling (where each loop has an unroll factor of 1), the model's predicted unrolling, and the gold standard we had established.

IV. Results

We compared our original comprehensive feature set against a more focused set of instruction count features. The superior performance of our original feature set underscores the complexity of the loop unrolling decision. While instruction counts provide a basic measure of loop complexity, they fall short, with just 58.65% accuracy as shown in Table II, in capturing the full range of factors that influence optimal unrolling.

This shows dependency analysis is fundamental to effective loop unrolling, as it enables compilers and optimizers to make informed decisions about how to transform loops, ensuring correctness and not overspeculating how many loop unrolls will still maintain the intended effect of the code. This is all while maximizing performance benefits through increased parallelism and improved instruction scheduling. From this experiment, we realized that testing our model using the first and original set of features was the best set of features to extract from loop examples and run our experiments.

TABLE II: Test Metric Values

Test Metric	Value
Accuracy	0.5865
Precision	0.5417
Recall	0.5865
F1 Score	0.5555

Figure 3 depicts a runtime comparison from a subset of the PolyBench suite. The left axis represents the difference between the measured runtime for the algorithm with unrolling (either none or that determined by the model) compared to our gold standard at 0. Ideally, we would expect to see a positive light blue bar and a minimal dark blue bar, which would represent our model achieving at or near the gold standard. In cases where the bars are even (ie. bigc, atax), we see no speedup which is indicative of our model finding an unroll factor of 1 for all loops for that benchmark. Bars with negative values (ie. gemver, cholesky) indicate that the runtime for that set of unroll factors was faster than that of the gold standard.

The right axis represents the percent speedup of our model's outputs compared to no unrolling. In cases where the model predicted no unrolling, the speedup is 0. In the cases where the dark blue bar depicts a lower value than the light blue, speedup is positive, indicating the model predicted a set of unroll factors that was more performant than not unrolling at all. Conversely, the speedup percentage dips below zero when the model predicts a set of unroll factors that is less performant than no unrolling (ie. cholesky).

Figure 4 shows the accuracy of model prediction compared to the predetermined gold standard unroll factors for each benchmark. The accuracy ranged from a high of 100% to a low of 20% with an average of 54%. It is important to note that this measure only refers to correctness of the model at selecting the single best unroll factor and does not consider whether or not the model selected an unroll factor that was still preferable to not unrolling.

V. Discussion

When approaching the problem of developing a predictive model to determine ideal loop unroll factors, we found a wide range of limitations that prevented us from having greater success. Perhaps the most critical was finding and labeling a dataset that was representative of cases with all different ideal unrolling factors. When determining the loop unroll factors to label our data, higher loop unroll factors were significantly less common. We noticed this problem fairly early on, trying a few different datasets before selecting the best. Even then, only 0.18% of loops had an optimal loop unroll factor of 8, while 2.61% of the loops received an optimal unroll factor of 6, 12.05% got 4, 27.97% got 2, and 57.19% got 1, which means no unrolling. With such an imbalanced dataset, it is difficult to prevent classifier bias towards our majority classes. One potential future direction would be to design a dataset with emphasis on ensuring an even distribution of loops with high and low unroll factors, rather than loops that are representative of most common code uses.

Additionally, when doing our evaluation, we found tremendous difficulty getting reliable runtimes throughout our experimentation. Initially, our approach would yield disparities ranging from negligible to over 2x. After a significant refactor, we managed to reduce that noise but still found that it interfered with our data collection. As shown in Figure 3, we had cases where our 'gold standard,' which was determined from data collected on the same hardware as the evaluation, was outperformed by collections of other unroll factors. This makes it hard to determine the reliability of data labeling, which was determined the same way as this 'gold standard,' as well as the results. This noise may have been mitigated by implementation of smoothing algorithms over more data, however the runtime of a single pass of data collection was too significant to run several passes within the timeline of course deadlines.

Our team opted to evaluate the model on PolyBench [7] to gauge how well this technique could be applied to real-world compilation applications. While our average accuracy was only 54%, we can note anecdotally that we did see improvements to runtime compared to no unrolling for around half of the benchmarks. It is notable that the 'gold standard' for PolyBench only contained 4 instances where the ideal loop unroll factor was 6 and none where the factor was 8, from over 300 loops. This is similar to our dataset, making it difficult to discern to what degree our model was biased towards lower unroll factors rather than predicting well. Another opportunity for future work would be to evaluate our model on other widely-accepted benchmarks, such as MiBench, to better gauge how it performs.

VI. Conclusion

In this paper, we introduced OpLUn, which aimed at addressing a critical decision problem compilers face: selecting the optimal loop unrolling factor. Through leveraging the MLP model, it is clear that data-driven approaches can substantially improve outcomes compared to traditional heuristics. Our evaluation on an array of different test cases as well as PolyBench verified the potential for ML-based approaches.

However, the results highlighted limitations particularly in scalability and the balance of the dataset. The model's accuracy and performance were set back by the skew towards choosing a lower loop unrolling factor, reducing the efficacy in more complex scenarios. Additionally, noise in measuring runtimes emphasized the challenges we faced. From these challenges comes the crux of future directions for the project: curating a more comprehensive dataset to cover a broader range of complexity in loops and contexts will be essential to better emulate real-world applications, and sufficiently improve scalability. Despite these minor drawbacks, the underlying potential of the approach remains unchanged and gives way for more efficient compiler optimization.

VII. Appendix

Features 1

Features:

- Nesting Level
- Number of Operations
- Number of Floating-Point Operations
- Number of Branch Instructions
- Number of Memory Operations
- Number of Operands
- Number of Implicit Instructions
- Number of Unique Predicates
- Critical Path Latency
- Cycle Length Estimate
- Programming Language
- Number of Parallel Computations
- Maximum Dependence Height
- Maximum Memory Dependency Height
- Maximum Control Dependency Height
- Average Dependence Height
- Number of Indirect References
- Minimum Memory-to-Memory Loop-Carried Dependency
- Number of Memory-to-Memory Dependencies
- Trip Count
- Number of Uses
- Number of Definitions

Features 2

Features:

- Number of Load Instructions
- Number of Store Instructions
- Number of Integer Compare Instructions
- Number of Branch Instructions
- Number of PHI Instructions
- Number of Add Instructions
- Number of Subtract Instructions
- Number of Multiply Instructions
- Number of Divide Instructions
- Number of Remainder Instructions
- Number of Memory Allocation Instructions
- Number of AND Instructions
- Number of OR Instructions
- Number of Select Instructions
- Number of Call Instructions
- Number of Switch Instructions
- Number of Number of BitCast Instructions
- Number of Truncation Instructions
- Number of Zero Extension Instructions
- Number of Float Multiply Instructions
- Number of Float to Signed Int Instructions
- Number of Unreachable Instructions

References

[1] M. Stephenson and S. Amarasinghe, “Predicting unroll factors using supervised classification,” in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2005, pp. 123–134. [Online]. Available: https://doi.org/10.1109/CGO.2005.9

[2] A. Haj-Ali, N. K. Ahmed, T. Willke, 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 (CGO)*. IEEE, 2020, pp. 242–253. [Online]. Available: https://ieeexplore.ieee.org/document/9076760

[3] E. Alwan and R. A. Baity, “Optimizing program efficiency with loop unroll factor prediction,” *Information Sciences Letters*, vol. 12, no. 6, pp. 2207–2213, 2023. [Online]. Available: https://www.naturalspublishing.com/files/published/jw3u471n2zvl23.pdf

[4] I. Singh, S. K. Singh, R. Singh, and S. Kumar, “Efficient loop unrolling factor prediction algorithm using machine learning models,” in *2022 3rd International Conference for Emerging Technology (INCET)*. IEEE, 2022, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/9825092

[5] G. Zacharopoulos, A. Barbon, G. Ansaloni, and L. Pozzi, “Machine learning approach for loop unrolling factor prediction in high level synthesis,” in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 191–198. [Online]. Available: https://ieeexplore.ieee.org/document/8514335

[6] D. M. Ritchie, “The development of the C language,” in *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*, ser. HOPL-II. New York, NY, USA: Association for Computing Machinery, 1993, pp. 201–208, [Online]. [Online]. Available: https://doi.org/10.1145/154766.155580

[7] M. J. Reisinger, “Polybench/c benchmark suite,” 2014. [Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1