VLOS : Vectorized LLVM Optimization System

Special Topics in Data Science(005) : ML4Sys Term Project

Junghyun Lee, Jinwook Yang

CODE Lab, SNU Graduate School of Data Science

December 10, 2024

Abstract

A vast amount of previous work has demonstrated that no single fixed optimization sequence is optimal for all types of code. However, searching for the best set of optimizations for each code individually is a tedious and computationally expensive process. As a compromise, the -O3 flag is commonly used during compilation, as it applies a batch of standard optimization passes. While -O3 often provides promising performance improvements, not all passes within -O3 are beneficial for every code. In this work, we propose VLOS, a framework that clusters Intermediate Representations (IRs) of code requiring similar optimal optimization passes. To ensure a balance between generality and granularity, we use k-means clustering. For feature extraction, we present a novel approach that embeds LLVM IR code using BERT for contextual understanding and a Graph Autoencoder (GAE) to capture structural relationships within a computational flow graph. Our results demonstrate that codes within the same cluster share highly similar optimization needs, validating the effectiveness of our feature extraction method. Furthermore, when comparing the optimization sequences generated by VLOS to those from the -O3 flag, VLOS shows the potential to provide more efficient and tailored optimization sequences, leading to improved performance.

1 Introduction

Early research dating back more than 25 years [3] has already highlighted the necessity of customized optimization passes to fully optimize a piece of code. Over the years, various methods have been proposed to select optimization passes, including Genetic Algorithms [3, 9], Reinforcement Learning (RL) [8], and Deep Neural Networks (DNNs) [1]. However, despite the efficiency of these methodologies, the optimization search space remains extremely large, often requiring significant time and computational resources. As a result, it is impractical for users to search for the optimal set of optimization passes for every piece of code they wish to compile.

As a practical alternative, compilers offer the -O# option, where # represents the optimization level. These levels range from -O0 to -O3, with higher levels corresponding to more aggressive optimizations. At -O0, no optimizations are applied, prioritizing faster compilation times and ease of debugging. -O1 introduces basic optimizations to improve performance with minimal trade-offs in compilation overhead. -O2 enables more advanced optimizations, such as loop transformations and code motion, striking a balance between performance and compilation time. Finally, -O3 applies aggressive optimizations, including vectorization and loop unrolling, to maximize performance but at the cost of increased compilation time and potentially larger binary size.

While the -O3 option generally ensures notable performance improvements, not all optimization passes within -O3 are equally effective for all types of code. In a preliminary experiment on LLVM IR, we observed a significant variance in the number of passes executed for different codes, even when the same -O3 flag setting was used. In extreme cases, there was a difference of up to 25 in the number of executed passes; for instance, one code underwent 28 passes, while another executed only 3.

This motivated our project, which aims to classify code into groups that share highly similar optimization sequences as they are transformed into an optimized state. The core principle we emphasize is maintaining a balance between granularity and generality in the optimization sequences. To achieve this, we use clustering to group codes based on extracted features, ensuring that the granularity of our approach remains at a meaningful level.

A critical step in this project was determining how and at what level of code features should be extracted. With the rapid advancement of feature extraction and embedding techniques, this choice significantly impacts the effectiveness of the representation. Various approaches have been proposed in previous work, including extracting features directly from raw source code [1, 4], from schedule primitives [11], and from Intermediate Representations (IRs) [5, 6].

We chose to embed LLVM IR into vector representations for the following three reasons:

- LLVM IRs are a portable, high-level assembly language that supports a wide range of optimization transformations
- Compared to raw source code, LLVM IRs eliminate most stylistic choices and redundant code, providing a cleaner and more consistent representation.
- A novel embedding method IR2VEC [10] already exists, offering a solid baseline for comparison against our proposed feature vector.

First, we attempted to embed a set of LLVM IRs using IR2VEC [10], a novel, flow-aware vectorization method widely used in previous works. However, two significant problems emerged with this approach. The first issue was that IR2VEC required a meticulous setup of the LLVM IR; the model only worked error-free if the LLVM version and the format of the IR were fully compatible. The second, and perhaps more critical issue, was that the vector space generated by IR2VEC were overly sparse, making them unsuitable for clustering tasks. These challenges are discussed in more detail in Section 3.1.

The unexpected lack of robustness in IR2VEC highlighted the need for a more stable and reliable feature extraction method. Inspired by the approach used in MIREncoder [6], where multiple models were employed to extract features from IRs, we developed our solution. Specifically, we use BERT to obtain rich contextual embeddings of the IR code, combined with a Graph AutoEncoder (GAE) to extract structural relationships from the IR's computational flow graph. The two embeddings are concatenated to form a unified feature vector, ensuring that both the flow and structure of the IR are effectively captured.

Using the resulting embeddings of the LLVM IRs, we applied k-means clustering to generate groups. We then analyzed the similarity of optimization sequences within each group to evaluate the quality of the clusters.

Finally, we tested our framework by embedding previously unseen LLVM IRs into feature vectors and using them as inputs to the VLOS framework. After assigning the new IR to the appropriate cluster, we applied the optimization sequence associated with the cluster's centroid IR. Comparing the performance of this sequence to the standard -O3 flag, our results demonstrate that VLOS can provide a more efficient optimization sequence in certain cases.

To summarize, the main contributions of our project are as follows:

- We propose a novel method for embedding LLVM IRs that incorporates both contextual understanding (using BERT) and structural relationships (using GAE) within the code.
- The rich feature vectors generated by our approach enables the creation of meaningful clusters.
- Experimental results show that the optimization sequences identified by VLOS strike a balance between granularity and generality, outperforming the -O3 flag in some cases.

The rest of this report is organized as follows. Section 2 describes the data used in our project. Section 3 discusses the limitations of the IR2VEC approach and provides a detailed explanation of our new feature vector generation method. Section 4 presents an overview of the VLOS framework and analyzes the intra-cluster similarity of the generated groups. Section 5 reports our experimental results, and finally, Section 6 concludes the report with our findings and future directions.

2 Data

For our project, we obtained LLVM IRs directly from the ComPile dataset [7]. This dataset consists of 2.7TB of permissively-licensed source code written in C/C++, Rust, Swift, and Julia, which has been

compiled into LLVM IR. The IRs are provided in an unoptimized state, offering an ideal starting point to apply and test various optimization passes and observe their effects. We used a subset of the dataset, comprising 20,000 IRs for training and 1,000 IRs for testing.

3 Embedding LLVM IRs

In this section, we discuss the methods used to vectorize LLVM IRs and cluster the resulting vectors. Section 3.1 explains our attempt to apply IR2VEC and highlights its limitations, particularly its unsuitability for our task despite considerable effort. Section 3.2 introduces our novel approach, which leverages BERT and GAE models for effective feature extraction.

3.1 IR2VEC

As mentioned in the previous sections, IR2VEC has been frequently used in prior works to embed LLVM IR. It can be summarized as a program embedding framework that represents code as continuous vectors in a hierarchical manner. IR2VEC leverages representation learning to generate seed embeddings for IR entities, such as instructions, operands, and types, by modeling their relationships using a knowledge-graph-inspired approach with TransE. These seed embeddings are further combined with classical program analysis techniques to create symbolic and flow-aware encodings at the instruction, basic block, function, and program levels. The resulting embeddings are both language- and machine-independent, making them suitable for downstream compiler optimization tasks. When applied, IR2VEC transforms the code into a 300-dimensional embedding vector.

Revisiting the two problems stated in Section 1, we first address the issue of IR2VEC's applicability. Not all IRs in the ComPile dataset could be successfully converted into vectors using the IR2VEC framework. IR2VEC supports up to LLVM version 17, but since LLVM versions are updated frequently, incompatible expressions, such as the introduction of opaque pointers, often caused errors during processing. Additionally, IR2VEC requires a complete and well-formed LLVM IR, where all functions must be fully declared and defined. Even minor mismatches could easily lead to undesired results, such as zero vectors or NaNs, which highlighted IR2VEC's lack of robustness and its error-prone nature.

Despite this challenge, the problem was solvable, and we carefully processed the ComPile dataset to identify IRs that could be converted correctly. As a result, we collected 20,000 IRs, which will henceforth be referred to as the **experiment set** for this project.

After converting the IRs in the **experiment set** into 300-dimensional embeddings, we applied the k-means clustering method to generate clusters. K-means was chosen for its simplicity and computational efficiency in partitioning the embeddings into distinct clusters by minimizing intra-cluster variance.

To measure the similarity between the 300-dimensional embeddings, we used cosine similarity as the distance metric. Unlike Euclidean distance, cosine similarity focuses on the direction rather than the magnitude of vectors, making it particularly effective in high-dimensional spaces. By mitigating the influence of varying vector magnitudes, it ensures that clustering is based on the relative orientation of the vectors, which is essential for embeddings where relationships are primarily encoded in their directions. This approach enabled effective grouping of similar IRs based on their extracted features.

To determine the optimal number of clusters k, we employed the elbow method. This involved plotting the total cosine distance within clusters against various values of k and identifying the "elbow point" on the graph, where the decrease in total cosine distance starts to level off. The elbow point strikes a balance between clustering accuracy and model simplicity, typically indicating the optimal value of k.

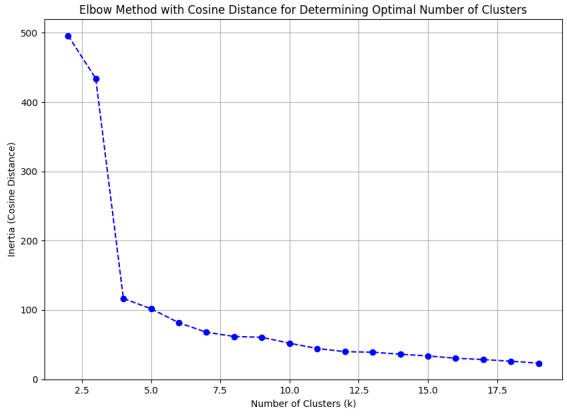

Figure 1: Finding the optimal k (IR2VEC)

Figure 1 shows the results of the elbow method. Under normal circumstances, k = 3 would have been selected as the optimal value. However, given the size and diversity of our experiment set, a higher value

of k, specifically 8, was chosen to better capture the underlying structure and variation within the data.

To visualize the clusters formed by the 300-dimensional embeddings, we applied t-SNE, a dimensionality reduction technique that projects high-dimensional data into a 2D space while preserving local structure and relationships between points. The resulting visualization is shown in Figure 2.

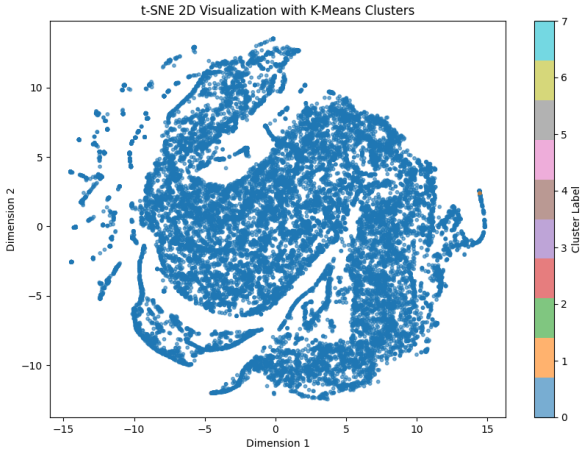

Figure 2: T-sne visualization of clustering (IR2VEC)

We expected that this process would allow us to observe the clustering patterns and assess the separability of the data in a more interpretable form. However, the results were quite disappointing. Nearly all IRs appeared to be grouped into a single cluster, with no points belonging to other clusters visible elsewhere—except for one orange point located toward the far right in Figure 2. Furthermore, when comparing the 50 nearest points to each centroid across the 8 clusters, we found that 48 points were common, meaning they overlapped across all clusters. This suggests that applying clustering to this data may not be the most appropriate approach, as the embeddings lack sufficient differentiation to form distinct clusters.

Upon further inspection of the vectors generated by IR2VEC, we identified the root cause of this issue. Many vectors contained values that were extremely large (or small) compared to others, leading to sparse and uneven distributions in the embedding space. This made it challenging for the clustering algorithm to form well-defined clusters. Furthermore, the extreme values distorted the distance relationships between vectors, resulting in uninformative t-SNE visualizations. This was the second, and ultimately unsolvable, problem with using IR2VEC, rendering it unsuitable for our task.

Not to mention, some lengthy IRs required over an hour for IR2VEC to generate embeddings for the entire code, making the embedding process highly time-consuming. This also poses a challenge for our system, particularly if the goal is to automate the entire process in a manner similar to a compiler.

3.2 BERT & GAE

At the start of the project, we anticipated two possible contributions regarding the embedding process. First, if clustering using the embedded vectors from IR2VEC proved successful, the goal of our newly proposed embedding method would be to outperform the vectors created by IR2VEC. Second, if IR2VEC turned out to be infeasible for some reason, the need for our proposed approach would be justified. Ultimately, it turned out to be the latter case.

Since IR2VEC is a flow-aware code representation that focuses on capturing program semantics but lacks structural awareness, we decided to adopt an alternative **dual approach**, combining BERT for embedding the code itself and a Graph Autoencoder (GAE) for computational graph representation. This approach improves feature extraction from LLVM IR by explicitly capturing structural relationships within the code, such as control and data dependencies, which are better represented in graph-based models. By combining BERT's ability to generate rich contextual embeddings with GAE's capability to preserve graph structure, this method captures both the semantic and structural aspects of the code. As a result, it provides a more comprehensive and robust representation, enabling better clustering and downstream analysis compared to IR2VEC alone. Additionally, the trained model provides a significant speed advantage over the time-consuming IR2VEC process by directly inferring the embedding vector.

The idea of using BERT for code encoding is not entirely new. In [6], BERT was shown to be an effective model for generating rich embeddings from code. A key insight from their work was the decision to train a BERT tokenizer from scratch on IRs, rather than using a pre-trained tokenizer trained on natural language. The reasoning behind this approach was that IRs contain fewer "words" than natural language and follow a more structured format.

Inspired by this idea, we adopted a similar process to integrate BERT into our project. Specifically, we utilized a BERT model with 6 hidden layers and 12 attention heads. To further improve tokenization efficiency and reduce the vocabulary size, we replaced

all numeric values in the LLVM IR with a generic "[NUM]" token. This allowed the tokenizer to focus on meaningful tokens while maintaining and operating effectively with a manageable vocabulary size of 20,000 words.

The BERT model has a limitation in processing code blocks due to its restricted token length, making it difficult to capture the full context of the code. To address this issue, we incorporated a Graph Autoencoder (GAE) with Masked Graph Autoencoding to embed the computational flow graph of LLVM IR, as introduced in [2]. Specifically, we generated .dot files from LLVM IR using LLVM's built-in dot-callgraph pass, which represents the graph structure through nodes and edges. By parsing these .dot files, we constructed sparse matrices that encoded the graph's nodes and edges, serving as inputs for GAE training.

For simplicity, we considered only the structural shape of the graph, excluding the node values. The GAE model comprises an encoder and a decoder; the encoder employs a dual-layer Graph Convolutional Network (GCN), while the decoder uses an inner-product mechanism with a sigmoid activation function. The model is trained by continuously encoding and decoding graph inputs, optimizing with a cross-entropy loss function to learn meaningful graph representations.

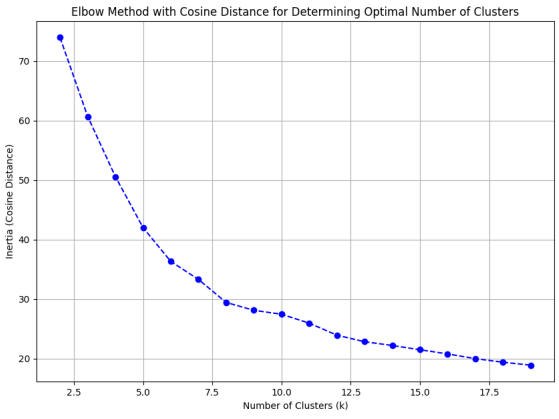

Figure 3: Finding the optimal k (Dual Embedding)

BERT produces a 768-dimensional embedding vector, while GAE generates a 32-dimensional embedding vector. The two outputs are concatenated to form a final 800-dimensional feature vector for each input IR. After embedding the **experiment set**, we applied the same process to these embeddings as we did for the IR2VEC embeddings in Section 3.1.

Figure 3 shows the results of the elbow method. Based on the observation of relatively small decreases in inertia beyond k=8, we selected k=8 as the optimal number of clusters. To visualize the clusters, we applied t-SNE, and the resulting visualization is shown in Figure 4.

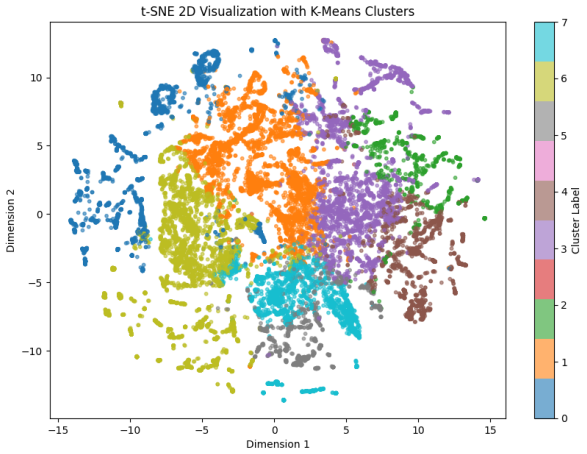

Figure 4: T-sne visualization of clustering (Dual Embedding)

Comparing the results of Figure 2 and Figure 4, the difference is immediately evident. The results are far more meaningful, with all 8 clusters clearly visible. This supports the effectiveness of our proposed method in generating meaningful, context-aware vectors for clustering.

4 VLOS

VLOS is a framework that embeds(vectorizes) an input IR and classifies it into the closest cluster. Within each cluster, all IRs are expected to share a similar list of optimization sequences.

4.1 Optimal Optimization Sequence

Defining the list of optimal optimization passes for a input LLVM IR was a critical component of this project. We restricted the candidate set of optimization passes to those included in the -O3 flag when compiling LLVM IR using clang (LLVM version 18). As mentioned earlier, not all passes in -O3 are always beneficial for every piece of code.

A total of 69 optimization passes are applied when the -O3 flag is enabled during the compilation process with LLVM version 18. To identify the passes that contribute to a reduction in instruction count, we implemented a script that checks and records the effect

of each pass. This was achieved using LLVM's -debug-pass-manager option, which generates logs detailing the impact of each optimization pass. From these logs, we extracted information on changes to the instruction count for every applied pass. The script then returned the passes that resulted in a reduction in instruction count.

When compiling all IRs in the **experiment set**, we observed an additional scenario: for code that did not benefit from the -O3 flag in terms of instruction count, a custom flag labeled NoPasses was returned.

In summary, we finalized a candidate set of 70 optimization passes, including the NoPasses option. For each IR, the **optimal optimization sequence** was defined as the sequence of passes that led to a reduction in instruction count.

4.2 Analyzing Generated Clusters

The goal of our project was to apply the **optimal optimization sequence** associated with the IR at the centroid of each cluster to all IRs assigned to the corresponding cluster. Using the explanation given in Section 4.1, this process is displayed in Figure 5.

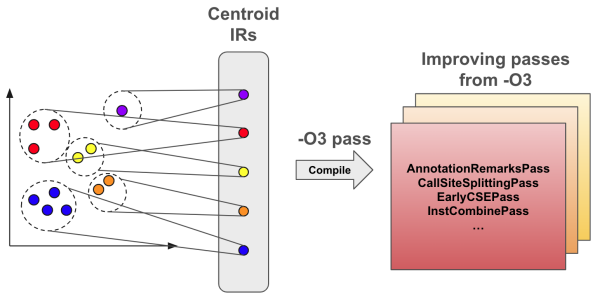

Figure 5: Process of defining OOS of Centroids

To evaluate the feasibility of using the centroid's **optimal optimization sequence** (hereafter referred to as the **OOS**), we selected the top-50 nearest embedded vectors (IRs) to each cluster's centroid. We then compared the similarity of their respective OOS with one another.

To compute the similarity, we represented each **OOS** as a binary 70-dimensional vector. If a pass was present in **OOS**, its corresponding index in the vector was set to 1; otherwise, the index was set to 0.

We used two metrics to assess similarity. The first was cosine similarity, a commonly used measure that evaluates the similarity between two vectors based on their orientation rather than their magnitude. Given two vectors A and B, the cosine similarity is calculated as follows:

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} \tag{1}$$

However, given that we are dealing with binary vectors, we decided to additionally compute the Tanimoto coefficient. The Tanimoto coefficient is a measure of similarity between two sets or binary vectors. It is a variation of the Jaccard index, specifically adapted for binary data, and accounts for the relative size of the sets. A value of 1 indicates perfect similarity, while a value of 0 indicates no similarity. The Tanimoto coefficient is calculated as follows:

$$\text{Tanimoto}(A, B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{2}$$

$$= \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sum_{i=1}^{n} A_i + \sum_{i=1}^{n} B_i - \sum_{i=1}^{n} A_i \cdot B_i} \tag{3}$$

The results showing the similarity of the **OOS** among the top-50 nearest IRs to each cluster's centroid are summarized in Table 1 using the two metrics.

Table 1: Average intra-cluster similarity using two metrics

Cluster	Cosine	Tanimoto
1	0.59	0.50
2	**0.90**	**0.90**
3	**0.95**	**0.92**
4	**0.90**	**0.82**
5	0.54	0.34
6	**0.74**	0.64
7	0.59	0.37
8	**0.81**	0.67

Out of the 8 clusters, 2, 3, and 4 show notably high similarity scores across both metrics, with values of 0.90 or higher, indicating strong consistency in the optimal optimization sequences (**OOS**) within these clusters. Cluster 3, in particular, achieves the highest scores with a Cosine similarity of 0.95 and a Tanimoto coefficient of 0.92, suggesting that the IRs in this cluster share highly similar optimization sequences.

Additionally, we compared the average length of the **OOS** for the top-50 nearest IRs to the length of the **OOS** of the cluster centroid. The purpose of this comparison was to evaluate similarity from a slightly different perspective: specifically, to determine whether the centroid's **OOS** could serve as a feasible representative **OOS** for its cluster. A small

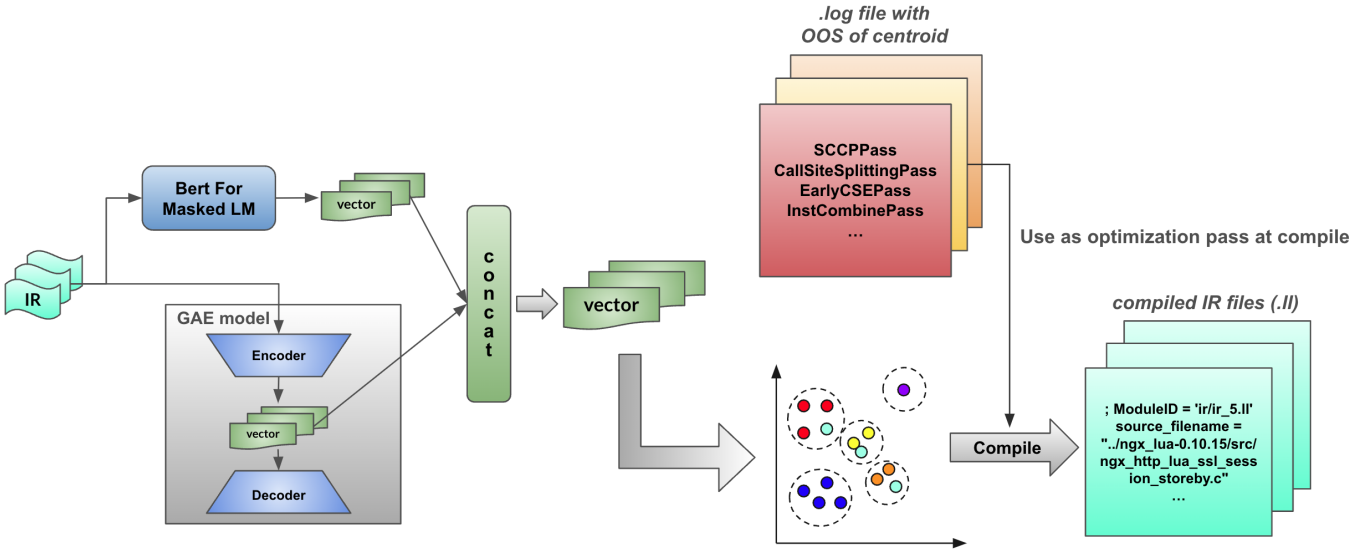

Figure 6: The Overview Framework of VLOS

difference between the two values would indicate that the centroid and the surrounding IRs form a reasonably well-defined cluster. The results are presented in Table 2.

Table 2: Comparison of OOS lengths

Cluster	Avg **OOS** len.	Centroid **OOS** len.
1	10.42	13
2	1.39	1
3	12.52	12
4	22.48	20
5	10.23	4
6	16.19	22
7	16.32	6
8	17.13	24

The results in Table 2 closely align with the findings in Table 1. For clusters with high similarity scores, the lengths of the centroid's **OOS** and the nearest vectors' **OOS** showed minimal differences. However, Clusters 5 and 7 produced slightly concerning results, raising reasonable doubts about whether these clusters are well-defined based on the extracted features.

Although a few clusters fell short of expectations on the similarity metric, the results generally demonstrate that certain clusters exhibit strong intra-cluster consistency, while others show greater variability. This variability may reflect differences in the characteristics of the IRs within those clusters or, more importantly, how the features are extracted using our proposed dual embedding method.

It is important to note that the intra-cluster similarity metric is not the final evaluation metric of our project. It serves as an intermediate step to assess the characteristics of the clusters. The evaluation of the efficiency of applying the centroid's **OOS** to an unseen, assigned IR is presented in Section 5.

4.3 Overview of Framework

To summarize all the components explained so far, the complete overview of the VLOS framework is illustrated in Figure 6. When an LLVM IR is provided as input, it is embedded into an 800-dimensional vector (comprising 768 dimensions from BERT and 32 dimensions from GAE, concatenated). The embedded vector is then assigned to the closest cluster. The **OOS** (Optimal Optimization Sequence) of the cluster's centroid is used as the list of optimization passes to be applied during the compilation process.

5 Results

After building the VLOS framework with the **experiment set**, we extracted 1000 additional LLVM IRs from the ComPile dataset to create a **test set**. These unseen IRs were transformed into feature vectors using the **dual embedding** method and were subsequently assigned to their closest clusters.

For each IR, the performance of the **OOS** provided by VLOS was compared to that of the -O3 flag. Two metrics were evaluated: instruction count and IPC (instructions per cycle). The results are presented in Table 3 and Table 4, respectively.

Table 3: Performance Evaluation (Instruction Count)

Cluster	Instruction Count				
	Average (Counts)			Improvement(%)	
	Before	VLOS	-O3	VLOS	-O3
1	874	769	804	**12.03**	8.00
2	1170	1170	1056	0	**9.75**
3	821	746	751	**9.15**	8.60
4	807	682	691	**15.45**	14.39
5	461	422	437	**8.35**	5.03
6	1033	890	877	13.85	**15.08**
7	1357	1061	1250	**21.78**	7.88
8	980	826	936	**15.74**	4.55

Table 4: Performance Evaluation (Instruction Per Cycle)

Cluster	Instruction Per Cycle				
	Average (IPC)			Improvement(%)	
	Before	VLOS	-O3	VLOS	-O3
1	1.20	1.10	1.21	-8.05	**1.23**
2	1.13	1.13	1.16	0	**3.23**
3	1.25	1.14	1.44	-8.21	**15.42**
4	1.18	1.06	1.13	-10.66	**-4.20**
5	1.00	0.88	1.00	-12.91	**-0.16**
6	1.16	1.02	1.13	-12.00	**-3.10**
7	1.31	1.27	1.43	-3.19	**9.11**
8	1.26	1.10	1.23	-12.40	**-2.60**

In terms of instruction count, VLOS generally demonstrates a competitive or superior reduction. VLOS shows notable improvements across several clusters, particularly in Cluster 7 (21.78%), Cluster 4 (15.45%), and Cluster 8 (15.74%) outperforming -O3 in reducing instructions. Except Cluster 2 and 6, VLOS overperformed -O3.

One particularly interesting result was observed in Cluster 2. The centroid's **OOS** for this cluster, as indicated in Table 2, contains only a single sequence: "NoPasses". In other words, Cluster 2 was expected to consist of IRs that would not benefit from any optimization passes. The high similarity scores computed in Table 1 for both metrics further suggested that this cluster was well-defined.

However, contrary to our initial assumption, the IRs in Cluster 2 did show improvements when the -O3 flag was applied. To investigate this slight discrepancy further, we plan to conduct additional experiments by testing individual optimization passes rather than using "NoPasses" as the **OOS** for Cluster 2.

That said, it is important to note that in scenarios where maximal optimization is absolutely critical (even at the cost of increased compile time), the results for Cluster 2 may not be adequate. However, in cases where a balance between compilation time and optimization level is acceptable, using no optimization passes for the codes assigned to Cluster 2 could be a viable and efficient solution.

Returning to Table 1 and Table 2, although Clusters 5 and 7 exhibit lower similarity scores, with Tanimoto values dropping to 0.34 and 0.37, respectively, and a significant difference in **OOS** length between the centroid and the top-50 nearest IRs, the results demonstrate that using the **OOS** from VLOS outperformed the -O3 flag. A much larger reduction in instruction count was achieved in these clusters.

Conversely, the IPC (Instructions Per Cycle) results

reveal a different trend. IPC is a performance metric that measures how many instructions a processor executes per clock cycle. It serves as a key indicator of how efficiently a program utilizes the CPU's execution resources. Higher IPC values generally imply better processor utilization and faster program execution, as more instructions are completed within each cycle.

The -O3 flag consistently outperforms VLOS in improving IPC across most clusters. Notable examples include Cluster 3 (15.42%), Cluster 7 (9.11%), and Cluster 1 (1.23%), where -O3 shows positive improvements, while VLOS results in negative IPC changes. This suggests that although VLOS effectively reduces instruction count, it does not always translate into improved execution efficiency as measured by IPC.

It is generally observed that IPC improves with the use of -O3 because it reduces unnecessary overhead, enables parallel execution through vectorization, and optimizes instruction flow. These optimizations allow the CPU to process more instructions in a single cycle, thereby improving overall efficiency and reducing execution time. However, since VLOS aims to produce a generalized optimal sequence for clusters of IRs, IPC may not always be the most appropriate or definitive metric for evaluating its effectiveness.

For the negative IPC values observed with -O3, we conclude the following two reasons for this outcome. First, aggressive optimizations such as loop unrolling or instruction reordering can sometimes negatively impact IPC by introducing unintended side effects, such as increased cache misses or pipeline stalls. Second, it is important to consider the variance in the results. Since the dataset we used was not specific to any particular domain, the outcomes varied significantly—some IRs showed substantial improvements with -O3, while others experienced significant decreases. Therefore, the negative average IPC does not imply that -O3 is inherently ineffective or harmful.

Overall, based on the results in Table 3, we conclude that VLOS provides an optimization sequence that effectively balances generality and granularity, as evidenced by significant decreases in instruction count.

Regarding compilation time, using VLOS is generally faster than -O3 because the optimization sequences it generates are shorter, showing 45% of improvement. Even when factoring in the inference time of BERT, the total compilation time remains comparable to that of using the -O3 flag, where using -O3 flag was 25% faster than VLOS.

6 Conclusion & Future Work

We proposed VLOS, a vectorized LLVM optimization system designed for IR feature extraction and optimization pass search. VLOS provides two key contributions to LLVM optimization. First, by utilizing dual embeddings from BERT and GAE models, our system improves the clustering of IR embeddings, enhances inference speed, and ensures compatibility across different LLVM versions. Second, we demonstrated that the cluster centroids can effectively identify optimal optimization sequences for nearby vectors, enabling efficient optimization and code size reduction for similar IRs. These results highlight the potential of vectorized LLVM IRs to simplify and streamline the search for optimization passes, paving the way for more efficient and automated pass sequence discovery.

Our project does, however, have certain limitations. Constrained by the project timeline, we employed a lightweight BERT model with a tokenizer limited to 15,000 tokens and a simplified graph autoencoder with only two convolutional layers. Utilizing larger models with more parameters could improve the accuracy of the embeddings. Additionally, an ablation study could be conducted to investigate the impact of varying embedding vector dimensions or adjusting the value of k in k-means clustering. Adding more IR data on the experiment set could lead to better findings of centroids. Furthermore, instead of relying solely on the k-means centroid for optimization passes, we could explore on finding better representative optimal optimization sequence for a cluster.

Lastly, the embeddings generated in VLOS could be applied beyond similarity analysis to other tasks, such as LLVM code generation, expanding the scope and versatility of the framework.

References

[1] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman P. Amarasinghe. A deep learning based cost model for automatic code optimization. *CoRR*, abs/2104.04955, 2021.

[2] Federico Cichetti, Emanuele Parisi, Andrea Acquaviva, and Francesco Barchi. Deepcodegraph: A language model for compile-time resource optimization using masked graph autoencoders. In Amon Rapp, Luigi Di Caro, Farid Meziane, and Vijayan Sugumaran, editors, *Natural Language*

Processing and Information Systems, pages 470–484, Cham, 2024. Springer Nature Switzerland.

[3] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, page 1–9, New York, NY, USA, 1999. Association for Computing Machinery.

[4] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[5] Akash Dutta, Jordi Alcaraz, Ali TehraniJamsaz, Eduardo Cesar, Anna Sikora, and Ali Jannesari. Performance optimization using multimodal modeling and heterogeneous gnn. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 45–57, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Akash Dutta and Ali Jannesari. Mirencoder: Multi-modal ir-based pretrained embeddings for performance optimizations, 2024.

[7] Aiden Grossman, Ludger Paehler, Konstantinos Parasyris, Tal Ben-Nun, Jacob Hegna, William Moses, Jose M Monsalve Diaz, Mircea Trofin, and Johannes Doerfert. Compile: A large ir dataset from production sources, 2024.

[8] Qijing Huang, Ameer Haj-Ali, William S. Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Juggling HLS phase orderings in random forests with deep reinforcement learning. *CoRR*, abs/2003.00671, 2020.

[9] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 171–182, New York, NY, USA, 2004. Association for Computing Machinery.

[10] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Transactions on Architecture and Code Optimization*, 17(4):1–27, December 2020.

[11] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning, 2022.