Learning Compiler Pass Orders using Coreset and Normalized Value Prediction

Youwei Liang[*1] **Kevin Stone**[*2] **Ali Shameli**[2] **Chris Cummins**[2] **Mostafa Elhoushi**[2] **Jiadong Guo**[2]
Benoit Steiner[3] **Xiaomeng Yang**[2] **Pengtao Xie**[1] **Hugh Leather**[2] **Yuandong Tian**[2]

Abstract

Finding the optimal pass sequence of compilation can lead to a significant reduction in program size. Prior works on compilation pass ordering have two major drawbacks. They either require an excessive budget (in terms of the number of compilation passes) at compile time or fail to generalize to unseen programs. In this work, instead of predicting passes sequentially, we directly learn a policy on the *pass sequence space*, which outperforms the default `-Oz` flag by an average of 4.5% over a large collection (4683) of unseen code repositories from diverse domains across 14 datasets. To achieve this, we first identify a small set (termed coreset) of pass sequences that generally optimize the size of most programs. Then, a policy is learned to pick the optimal sequences by predicting the normalized values of the pass sequences in the coreset. Our results demonstrate that existing human-designed compiler passes can be improved with a simple yet effective technique that leverages pass sequence space which contains dense rewards, while approaches operating on the individual pass space may suffer from issues of sparse reward, and do not generalize well to held-out programs from different domains. Website: https://rlcompopt.github.io.

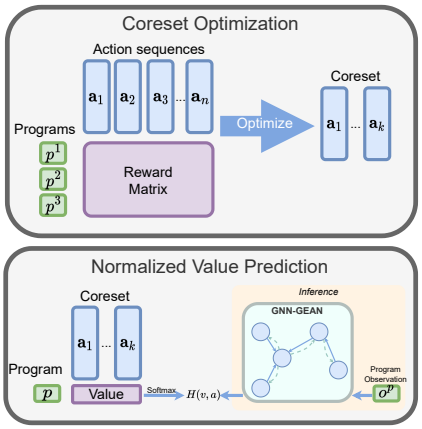

Figure 1. A depiction of our main contributions. (Top) **Coreset Optimization**: A process for discovering a small set of pass sequences (*coreset*) that generalizes. (Bottom) **Normalized Value Prediction**: A process where our model learns to predict the normalized value of pass sequences from the coreset.

1. Introduction

For more efficient execution with fewer resources (e.g., memory, CPU, and storage), applying the right ordering for compiler optimization passes to a given program, i.e., *pass ordering*, is an important yet challenging problem. Manual efforts require expert knowledge and are time-consuming, error-prone, and often yield sub-par results, due to the huge size of the search space. For example, the LLVM compiler has 124 different compilation passes. If the pass sequences have a length of 45, then the possible number of sequences ($124^{45} \sim 10^{94}$) is already more than the atoms in the universe ($\sim 10^{80}$ (Planck Collaboration et al., 2016)).

To address this problem, optimization-based approaches (e.g., MLGO (Trofin et al., 2021), MLGoPerf (Ashouri et al., 2022)) run adaptive search algorithms to optimize a set of programs for many hours. While this achieves strong performance gain, the procedure can be slow and does not distill knowledge from past experience and requires searching from scratch for unseen programs.

Recently, machine learning (ML)-guided pass ordering has emerged as an interesting field to replace this laborious process (Wang & O'Boyle, 2018). Along this line, many works show promising results using language modelling (Cummins et al., 2017), evolutionary algorithms (Kulkarni & Cavazos, 2012), and reinforcement learning (Haj-Ali et al., 2020a) to achieve better specific ordering for given programs. To handle unseen programs, Autophase (Haj-Ali

[*]Equal contribution, listed alphabetically. [1]University of California, San Diego [2]Meta AI [3]Anthropic. Correspondence to: Kevin Stone <kevinleestone@meta.com>, Yuandong Tian <yuandong@meta.com>.

Proceedings of the 40^{th} International Conference on Machine Learning, Honolulu, Hawaii, USA. PMLR 202, 2023.

et al., 2020b) learns a *pass selection policy* via reinforcement learning, and applies it to unseen programs without further search procedure, and GO (Zhou et al., 2020) finetunes the models for unseen programs. While these approaches show promosing results for unseen programs from the same/similar domain, they can be quite slow in the training stage, and have not shown good generalization to programs from very different domains, due to the fact that for different programs, the benefits are manifested only after a cleverly designed long pass ordering path (i.e., *sparse reward* in reinforcement learning).

In such cases, applying sequential search techniques, even guided with SoTA machine learning techniques, can be ineffective and may lead to overfitting of local noisy reward signals. To deal with these issues, we propose a novel pass ordering optimization pipeline to reduce the *code size* of a program. The key idea is the following: instead of searching good passes sequentially, we directly find a *universal* core set of pass sequences (termed **coreset**) from the training set, and make decision on top of this *different action space* to avoid the challenge of sparse reward.

Construction of the coreset. Specifically, the coreset is constructed by approximately optimizing a submodular objective with a greedy approach that has approximation guarantees. The resulting coreset contains 50 pass sequences of varying lengths, each of which has an average number of 12.5 passes. Surprisingly, despite the huge search space of compiler passes, the small coreset gives strong performance across programs from diverse domains, ranging from the Linux Kernel to BLAS. Specifically, the 50 pass sequences in the coreset lead to an average code size reduction of 5.8% compared to the default `-Oz` setting, across 10 diverse codebases (e.g. Cbench (Fursin, 2014), MiBench (Guthaus et al., 2001), NPB (Bailey et al., 1995), CHStone (Hara et al., 2008), and Anghabench (Da Silva et al., 2021)) of over one million programs in total.

Picking good pass sequences from the coreset. While it is still time-consuming to find an optimal pass sequence from the coreset with exhaustive search, we find that the (near) optimal pass sequence can be directly predicted with high accuracy via a graph neural network (GNN) architecture adapted to encode the augmented ProGraML (Cummins et al., 2021) graphs of programs. Therefore, we can run a few pass sequences selected by the model on an unseen program to obtain a good code size reduction. This enables us to find a good pass configuration that leads to 4.5% improvement on average, with just 45 compilation passes, a reasonable trade-off between the cost of trying compilation passes and the resulting performance gain.

We compare our approach with extensive baselines, including reinforcement learning (RL) -based methods such as PPO, Q-learning, and behavior cloning. We find that RL-based approaches operating on the original compiler pass space often suffer from unstable training (due to inaccurate value estimation) and sparse reward. As a result, they fail to generalize to unseen programs at inference. In comparison, our approach transforms the vast action space into a smaller one with much more densely distributed rewards. In this transformed space, approaches as simple as behavior cloning can be effective and generalizable to unseen programs.

2. Related Work

Recently, many methods have been proposed to use deep learning to perform compiler optimization.

CodeBERT (Feng et al., 2020) pre-trained language models on program languages with different pre-training designs, and finetuned the pre-trained models in downstream tasks including code document generation and code search. GraphCodeBERT (Guo et al., 2020) further extended CodeBERT to leverage data flows in pre-training with more pre-training objectives such as edge prediction and node alignment. Their main contribution is learning program representations, while our main contribution is compiler optimization with a manipulated search space. Their representation is at the source code level, while our representation is at the Intermediate Representation (IR) level.

Cereda et al. (2020) used similarity matching, an idea from recommender systems, to select optimization passes for programs. However, they only considered 7 optimization passes and only considered whether to apply them without considering their orders (thus making it a binary decision for each pass). Moreover, their approach needs to compare an input program with all the programs in the dataset for similarity matching, which may not scale up to a large dataset consisting of thousands of programs. Note that the inference overhead of our approach is insensitive to the size of the training set.

Mammadli et al. (2020) aimed for program *runtime* reduction via pass ordering, while we aim for program size reduction. Moreover, they used a tiny dataset - 109 single-source benchmarks from an LLVM test suite, while we use a much larger dataset set. The small size of their datasets and the lack of test sets may make their method prone to overfitting. The core learning algorithm in Mammadli et al. (2020) is deep Q learning (Mnih et al., 2015). We compared our method against another reinforcement learning (RL) algorithm PPO (Schulman et al., 2017).

Mammadli et al. (2021) proposed LoopLearner, which, given the source code of a loop, suggests a semantically invariant transformation that will likely allow the compiler to produce more efficient code. Their method applies only to loops in programs for program speedup.

Autophase (Haj-Ali et al., 2020b) proposed to extract statistics of IR instructions and use PPO to find a sequence of compilation passes that minimizes program execution time. They used 100 randomly generated programs as the training set and nine real benchmark programs as the test set, while we used a much large real-world training set and test set. MLGO (Trofin et al., 2021) integrated machine learning techniques, including policy gradient and evolution strategies, in an industrial compiler LLVM, which used the inlining pass to reduce program size, while we use a much larger set of compiler passes for program size reduction. MLGoPerf (Ashouri et al., 2022) used the inlining pass to optimize program speed with reinforcement learning.

Zhou et al. (2020) proposed the GO framework, targeting the optimization of the compilers for computational graphs in deep learning. Brauckmann et al. (2020) used abstract syntax trees and control flow graphs for learning compiler optimization goals. They show that using such graphs allows them to outperform state-of-the-art in the task of heterogeneous OpenCL mapping.

Cummins et al. (2021) proposed a graph-based representation for IR, called ProGraML, encoding the data flows, control flows, and function calls in programs. Cummins et al. (2022) provided a Python library CompilerGym for compiler optimization, supporting the construction of various program features and convenient interactions with compilers. Our work is based on both ProGraML and CompilerGym.

3. Method

3.1. Action space

The CompilerGym framework (Cummins et al., 2022) provides a convenient interface for the compiler pass ordering problem. The default environment allows choosing one of 124 discrete actions at each step corresponding to running a specific compiler pass. In this work we will use the term pass interchangeably with action. We fix the maximum number of passes to compile a program to 45 to match the setup in Haj-Ali et al. (2020b); Cummins et al. (2022). Given that our trajectories have a length of 45 steps, this means we have $124^{45} \sim 1.6 \times 10^{94}$ possible pass sequences to explore. To find an optimal pass sequence for a program, we can apply some existing reinforcement learning methods including Q learning like DQN (Mnih et al., 2015) and policy gradient like PPO (Schulman et al., 2017).

Pass Sequences. However for this problem it turns out that certain pass sequences are good at optimizing many different programs (where "good" is defined as better than the compiler default `-Oz`). We found that constraining the action space to a learned set of pass sequences enables state of the art performance and also significantly reduces the challenge of exploration. This allows us to cast the problem as one of supervised learning over this set of pass sequences. We use the following algorithm to find a good set of pass sequences.

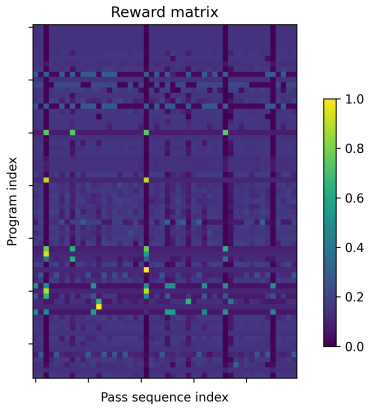

Figure 2. An exemplar reward matrix for 67 programs and 50 pass sequences. The values plotted are the pre-normalized values. Most of the pass sequences do not lead to strong rewards, except for a few. On the other hand, certain pass sequences (i.e., columns) can lead to high rewards for multiple programs simultaneously and thus are good candidates for the coreset.

Suppose we have N programs and M promising pass sequences. Let $R = [r_{ij}] \in \mathbb{R}^{N \times M}$ be the reward matrix, in which $r_{ij} > 0$ is the *ratio* of the codesize of i-th program if applied with j-th pass sequence, compared to `-O0` (i.e., the code size without compiler optimization). $r_{ij} > 1$ means that the j-th pass sequence does *better* than `-O0` in codesize reduction for i-th program, and $r_{ij} < 1$ means it performs worse. The reward matrix is normalized per row, by the maximum reward for each program, so that the optimal pass sequence has reward of 1 for each program.

Then we aim to pick a subset S of K pass sequences, called the *coreset*, from all M pass sequences, so that the overall saving $J(S)$ is maximized:

$$\max_{|S| \le K} J(S) = \sum_{i=1}^{N} \max_{j \in S} r_{ij} \tag{1}$$

Finding M candidate pass sequences. Note that there can be an exponential number of pass sequences, and we cannot construct the entire reward matrix, instead we seed a list of candidate pass sequences. For this, we run a random policy on a subset of M (17500) selected training programs. In applying the random policy, we uniformly sample a sequence of 45 passes in each episode, run E (200) episodes on a program, and pick the best pass sequence as the candidate sequence of the program, resulting in M candidate sequences in total. If part of the best pass sequence leads to the same state (the state is a 40-digit SHA1 checksum of

the program's IR, which can be obtained from the `IrSha1` observation in CompilerGym), they are truncated so that the sequence becomes shorter. If multiple pass sequences yield the same reward we only retain the first after ordering them length-lexicographically. On average these last two steps reduce the length of the candidate pass sequences by 80%. We then construct the reward r_{ij} by applying the j-th pass sequence to program i, and comparing it with `-O0`.

Finding the best coreset S with a greedy algorithm. As a function defined in subsets, $J(S)$ can be proven to be a nonnegative and monotone submodular function (See Appendix). While maximizing a submodular function is NP-hard (Ward & Živnỳ, 2016), the following *greedy algorithm* is proven to be an efficient approximate algorithm that leads to fairly good solutions (Nemhauser et al., 1978). Starting from $S_0 = \emptyset$, at each iteration t, it picks a new pass sequence j_t as follows:

$$j_t := \arg \max_{j \notin S_{t-1}} J(S_{t-1} \cup \{j\}) \tag{2}$$

And $S_t \leftarrow S_{t-1} \cup \{j_t\}$ until we pick K pass sequences. We set $K = 50$ in this paper as a larger coreset showed diminishing improvement (a coreset with 50 sequences already accounted for 95% of the improvement of using all M candidate sequences).

Given the discovered coreset S, we define a **generalized action** as a pass sequence in S. Applying a generalized action to a program means that we roll out the corresponding pass sequence on the program and return the best program state (i.e., having the highest cumulative reward), which can be done by caching the program state at each step.

Obtaining this coreset took 641 CPU core-hours. Less than 1% of time was spent on minimizing the size of the coreset. Most of the time was spent on random exploration on the training set to generate a promising set of pass sequences. Generating the coreset is an upfront cost that is paid only once. What was surprising to us is how well this generalized to unseen programs (as can be seen in the performance of our method on the held-out test set in Table 3).

3.2. Normalized Value Prediction

After discovering the "good" pass sequences (i.e., the coreset), we can turn the problem of the sequential decision-making on compiler passes into a problem of supervised learning. We aim to train a model to predict the best pass sequence conditioned on the program, where the training target is the index of the pass sequence that results in the greatest code size reduction. However, one important observation we have is that there are typically multiple pass sequences in the coreset that lead to the greatest code size reduction (see Figure 3 for the examples). Therefore, instead of predicting a single class label, we leverage the fact that we have access to the values for all pass sequences and predict the softmax normalized values of the pass sequences detailed below. This approach is similar to behavior cloning (Pomerleau, 1988) but with soft targets over the coreset.

For a program with an index i, we use r_{ij} to denote the reward (i.e., the code size reduction) when it is applied with j-th sequence, which forms a value vector $\boldsymbol{r}_i = [r_{ij}]_{j=1}^K$. Then, the normalized values of the pass sequences are defined by

$$\boldsymbol{v}_i = \text{Softmax}(\boldsymbol{r}_i / T) \tag{3}$$

where T is a temperature parameter.

For an initial observation o_i of the program, our model outputs a probability distribution, $\boldsymbol{a}_i = f(o_i)$, over the pass sequences. The target of the training is to make $\boldsymbol{a}_i$ close to the normalized values of the pass sequences. To this end, we use the Kullback–Leibler (KL) divergence to supervise the model, which can be reduced to the following cross entropy loss up to a constant term.

$$\mathcal{L}(\boldsymbol{v}_i, \boldsymbol{a}_i) = -\sum_{j=1}^{K} v_{ij} \log a_{ij} \tag{4}$$

3.3. Program Representations

Since we use the CompilerGym (Cummins et al., 2022) environments for program optimization, we exploit the program representations from CompilerGym, where program source code is converted to LLVM IR (Lattner & Adve, 2004) and several representations are constructed from the IR, including the ProGraML graph (Cummins et al., 2021), the Autophase feature (Haj-Ali et al., 2020b), and the Inst2vec feature (Ben-Nun et al., 2018).

Autophase We use the Autophase features (Haj-Ali et al., 2020b) to build some baseline models, which will be detailed in Section 4.2. The Autophase feature is a 56-dimension integer feature vector summarizing the LLVM IR representation, and it contains integer counts of various program properties such as maximum loop depth. we use an MLP to encode it and output a program representation.

ProGraML In addition to the Autophase features, we also leverage ProGraML (Cummins et al., 2021) graphs for training GNN models. ProGraML is a graph-based representation that encodes semantic information of the program which includes control flow, data flow, and function call flow. This representation has the advantage that it is not a fixed size - it does not oversimplify large programs - and yet it is still a more compact format than the original IR format. Each node in a ProGraML graph has 4 features described in Table 1. The "text" feature is a textual representation and the main feature that captures the semantics of a node.

For example, it tells us what an "instruction" node does (e.g., it can be `alloca`, `store`, `add`, etc). Each edge in a ProGraML graph has 2 features described in Table 1.

Extending ProGraML with type graphs There is an issue with the ProGraML graph. Specifically, a node of type variable/constant node can end up with a long textual representation (for the "text" feature) if it is a composite data structure. For example, a struct (as in C/C++) containing dozens of data members needs to include all the members in its "text" feature. In other words, the current ProGraML representation does not automatically break down composite data types into their basic components. Since there is an unbounded number of possible structs, this prevents 100% vocabulary coverage on any IR with structs (or other composite types). To address this issue, we propose to expand the node representing a composite data type into a type graph. Specifically, a pointer node is expanded into this type graph: `[variable] <- [pointer] <- [pointed-type]`, where `[]` denotes a node and `<-` denotes an edge connection. A struct node is expanded into a type graph where all its members are represented by individual nodes (which may be further expanded into their components) and connected to a `struct` node. An array is expanded into this type graph: `[variable] <- [array] <- [element-type]`. The newly added nodes are categorized as `type` nodes and the edges connecting the type nodes are `type` edges. The type nodes and type edges constitute the type sub-graphs in the ProGraML graphs. In this manner, we break down the composite data structures into the type graphs that consist of only primitive data types such as `float` and `i32`.

3.4. Network Architecture

Since the Autophase feature can be encoded by a simple MLP, we discuss only the network architectures for encoding the ProGraML graphs in this section.

We use a graph neural network (GNN) as the backbone to encode the ProGraML graphs and output a graph-level representation. The GNN encodes the graph via multiple layers of message passing and outputs a graph-level representation by a global average pooling over the node features. The goal of graph encoding is to use the structure and relational dependencies of the graph to learn an embedding that allows us to learn a better policy. To this end, we experimented with several different GNN architectures such as Graph Convolutional Network (**GCN**) (Kipf & Welling, 2017), Gated Graph Convolutions Network (**GGC**) (Li et al., 2015), Graph Attention Network (**GAT**) (Brody et al., 2022), Graph Isomorphism Network (**GIN**) (Xu et al., 2019). To better capture the rich semantics of node/edge features in the ProGraML graphs, we propose Graph Edge Attention Network (**GEAN**), a variant of the graph attention network (Veličković et al., 2017). These GNNs leverage both the node and edge features, so we start by presenting how to embed the node and edge features.

	Feature	Description
Node	type	One of {instruction, variable, constant, **type**}
	text	Semantics of the node
	function	Function index
	block	IR basic block index
Edge	flow	Edge type. One of {call, control, data, **type**}
	position	Integer edge position in flow branching

Table 1. Features in the ProGraML graph representation which we augment with type information (changes highlighted). We ablate the augmentations in Section 4.5.

Node embedding For the "text" features of the nodes, we build a vocabulary that maps from text to integer. The vocabulary covers all the text fields of the nodes in the graphs in the training set. The final vocabulary consists of 117 unique textual representations, and we add an additional item "unknown" to the vocabulary which denotes any text features that may be encountered at inference time and we have not seen before. The i-th textual representation is embedded using a learnable vector $\mathbf{x}_i \in \mathbb{R}^d$, where d is the embedding dimension. The "type" feature is not used because it can be inferred from the "text" feature.

Edge embedding The edge embedding is the sum of three types of embedding as the following.

- **Type embedding** We have 4 types of edge flows, so we use 4 learnable vectors to represent them.
- **Position embedding** The "position" feature of an edge is a non-negative integer which does not have an upper bound. We truncate any edge positions larger than 32 to 32 and use a set of 32 learnable vectors to represent the edge positions.
- **Block embedding** We use the block indices of the two nodes connected by the edge to construct a new edge feature. The motivation is that whether the edge goes beyond an IR basic block can influence program optimization. Suppose the block indices of the source node and the target node of an edge are respectively b_i and b_j. We get the relative position of the two nodes with respect to IR basic blocks in the following way: $p_{block} = \text{sign}(b_i - b_j)$. If the edge connects two nodes in the same IR basic block, then p_{block} is 0. And $p_{block} = \pm 1$ indicates the edge goes from a block to the next/previous block. There are 3 possible values for p_{block}, so it is embedded using 3 learnable vectors.

The final embedding of an edge is the sum of its type, position, and block embedding vectors.

Graph mixup We note that the ProGraML graphs of two programs can be composed into a single graph without affecting the semantics of the two programs. And their value vectors can be added up to correctly represent the value vector of the composite graph. In this manner, we can enrich the input space to the GNNs and mitigate model overfitting for the normalized value prediction method.

Graph Edge Attention Network We introduce the GEAN in this paragraph and defer its mathematical details to the Appendix. There are two main differences between the GAT and GEAN. 1) GEAN adopts a dynamic edge representation. Specifically, GAT uses the node-edge-node feature to calculate the attention for neighborhood aggregation, while GEAN uses the node-edge-node feature to calculate not only the attention but also a new edge representation. Then, the updated edge representation is sent to the next layer for computation. Note that GAT uses the same edge embedding in each layer. We conduct an ablation study showing that the edge representation in GEAN improves the generalization of the model. 2) GAT treats the graph as an undirected graph while GEAN encodes the node-edge-node feature to output an updated node-edge-node feature, where the two updated node features represent the feature to be aggregated in the source node and the target node, respectively. This ensures that the directional information is preserved in the neighborhood aggregation.

3.5. Dataset Preparation

Overfitting issues could happen if training is performed on a small subset of programs, or the set of programs is not diverse enough. To mitigate this we find it helpful to create an aggregate dataset that uses many different public datasets as curated by CompilerGym, selecting the program benchmarks with a maximum of 10k IR instruction counts. CompilerGym gives us access to 14 different datasets constructed using two different methods, where a benchmark program in the datasets is a single translation unit (i.e. object file).

- `Curated` These are small collections of hand-picked programs. They are curated to be distinct from one another without overlap and are not useful for training. Typically programs are larger as they may comprise multiple source files combined into a single program. These are commonly used for evaluating compiler optimization improvements.

- `Uncurated` These are comprised of individual compiler IRs from building open source repositories such as Linux and Tensorflow. We also include synthetically generated programs, targeted for compiler testing (not optimization).

Type	Dataset	Train	Val	Test
Uncurated	anghabench-v1	707,000	1,000	2,000
	blas-v0	133	28	29
	github-v0	7,000	1,000	1,000
	linux-v0	4,906	1,000	1,000
	opencv-v0	149	32	32
	poj104-v1	7,000	1,000	1,000
	tensorflow-v0	415	89	90
	clgen-v0	697	149	150
	csmith-v0	222	48	48
	llvm-stress-v0	697	149	150
Curated	cbench-v1	0	0	11
	chstone-v0	0	0	12
	mibench-v1	0	0	40
	npb-v0	0	0	121
Total	-	728,219	4,495	4,683

Table 2. CompilerGym dataset types and training splits. The hand-curated datasets are used solely to evaluate generalization to real-world program domains at test time. The units of the numbers are "benchmarks" as in the CompilerGym, where a benchmark represents a particular program that is being compiled.

For our aggregate dataset we decided to holdout the entirety of the four curated datasets for use as an out-of-domain test set. This is important because they represent the types of programs we expect to see in the wild. We also split the uncurated datasets into train, validaton, and test programs.

We limited the size of the programs by setting the maximum IR instruction counts of a program to 10k for two reasons. First, we need to consider the GPU memory constraints. Second, it may not be optimal to embed a very large program like the Linux kernel to obtain a single embedding, for which our model will output a few pass sequences. This is because we want to perform fine-grained optimization over the program units because each part of a large program may react differently to optimization pass sequences.

3.6. Evaluation

For all our metrics and rewards we leverage the IR instruction count as value we are trying to minimize. We also report metrics on each CompilerGym dataset as well as the mean over datasets to get a single number to compare overall results.

- The mean percent improved over `-Oz` (**MeanOverOz**) (Haj-Ali et al., 2020b) defined as following:

$$\bar{I}^{Oz} = \text{MeanOverOz} := \frac{1}{|\mathcal{P}|} \sum_{p} \frac{I_p^{Oz} - I_p^{\pi_\theta}}{I_p^{Oz}}, \quad (5)$$

where p is a specific program from the set of programs $\mathcal{P}$ in the dataset. I_p^{Oz} is the number of IR instructions

in the program after running the default compiler pass `-Oz`. $I_p^{\pi_\theta}$ is the number of IR instructions in the program after applying the policy under consideration. We can think of this as a simple average of the percent improvement over `-Oz`.

- We also compare the geometric mean (**GMeanOverOz**) (Cummins et al., 2022) of final sizes across all programs relative to `-Oz` to give a weighted comparison that is insensitive to outliers.

$$\bar{I}_G^{Oz} = \text{GMeanOverOz} := \left(\prod_p \frac{I_p^{Oz}}{I_p^{\pi_\theta}} \right)^{\frac{1}{|\mathcal{P}|}} \quad (6)$$

4. Experiments

4.1. Experimental Setup

The experiments are based on the CompilerGym (Cummins et al., 2022) environment. Given the source code of a program, an optimization policy proposes a sequence of 45 compiler passes. Although the compiler can accept multiple passes in a single invocation, we apply the 45 passes with 45 invocations of the compiler in our setting so that we can cache the intermediate IR between passes. At each invocation, the program size (measured by the IR instruction counts) is recorded. The smallest program size during this process is used to calculate the performance metrics. For our method and the baseline methods, we search over a set of hyper-parameters, including the temperature T in Eq. 3, number of layers, the embedding dimension of the node/edge features, and the output dimension in the hidden layers in the MLPs. We select the best configuration for each method based on the validation metric (validation MeanOverOz in 45 steps). Then, we run the training and testing with the best configuration of a method for 3 times with different random seeds.

4.2. Baseline Methods

`-Oz` This is the Clang compiler's default optimization for program size reduction, and the passes sequence of `-Oz` is always the same for all programs. The number of passes applied by `-Oz` is 97, which are all exposed through CompilerGym as actions. `-Oz` also runs additional analyses that are not included in CompilerGym. Our approach is limited to 45 passes, using fewer passes then `-Oz`.

Oracle We consider a brute-force search over the coreset in order to find the best pass sequence for a given program. This gives us an upper-bound of the downstream policy network. In our case the coreset has 50 sequences and a sequence has an average number of 12.5 passes, resulting in a total of 625 passes in the coreset. The brute-force search is to roll out each sequence in the coreset and use the result from the best one.

Top-45 We also consider how well we would do if the oracle is only allowed to use the first few pass sequences in the coreset but limited to 45 passes. By the construction of the coreset, the first few sequences are the ones that are most popular. Any passes after the first 45 passes are truncated.

RL-PPO We reproduce the Autophase (Haj-Ali et al., 2020b) pipeline by using the state-of-the-art RL algorithm PPO (Schulman et al., 2017) to learn a policy model. We have two program representations for training the RL models, including the Autophase feature and the ProGraML graphs (note that Haj-Ali et al. (2020b) only used the Autophase feature). The Autophase/ProGraML feature is sent to a GNN/MLP for feature encoding, which outputs a program-level embedding. Following Haj-Ali et al. (2020b), we add an additional action history feature to the RL pipeline, which is a histogram of previously applied passes. The vector of the histogram of action history is divided by 45 (i.e., the number of the total passes in our budget) for normalization. A 2-layer MLP is used to encode the action history to obtain a feature vector, which is concatenated with the program embedding extracted from the ProGraML graph or the Autophase feature. The concatenated feature is sent to a 2-layer MLP to output the action probability for the policy network. The value network (i.e., the critic) in our PPO pipeline mimics the policy network (i.e., the actor) in feature encoding and outputs a scalar to estimate the state values. The state values are the expectation of the discounted cumulative rewards where the reward in each step is the improvement over `-O0`: $(I_p^{(t)} - I_p^{\pi_\theta})/I_p^{(t)}$, where $I_p^{(t)}$ denotes the current IR instruction count of the program p at time step t. This reward is reasonable since it makes the value approximation Markovian. At inference, an action is sampled from the output of the learned policy network at each time step until the total number of steps reaches 45.

Q-value-rank We consider each pass sequence in the coreset as a *generalized action* and train a Q network to predict the value of each generalized action. Recall that the value vector $\boldsymbol{r}^p$ is the highest cumulative reward observed during the rollout of each pass sequence in the coreset on program p. The Q-value-rank model is trained to approximate the value vector using a mean squared loss.

BC We consider learning a standard behavior cloning model to predict the best pass sequences from the coreset, where the best pass sequence is defined as the following. As in the previous paragraph, the value vector is denoted by $\boldsymbol{r}^p$. If there is only one i such that $r_i^p = \max_{j \in n} r_j^p$, then the classification label is i. If there are multiple such i's (multiple pass sequences) that achieve the largest reward $\max_{j \in n} r_j^p$, then we order the corresponding pass sequences by length-lexicographic ordering. The classification label is selected to be the first one after the ordering. This ensures that our

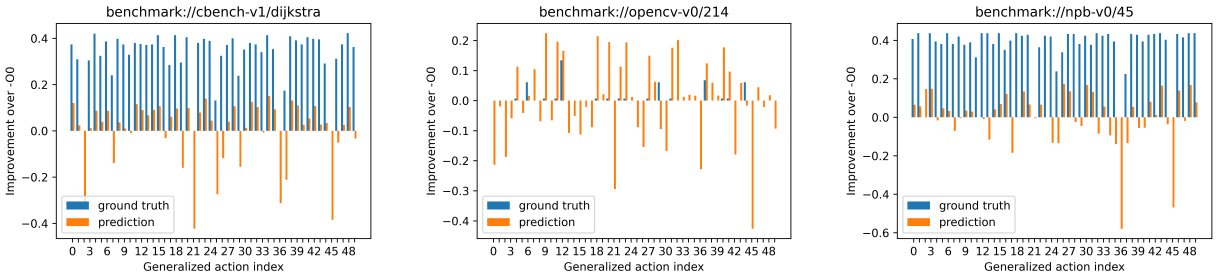

Figure 3. `GEAN-Q-value-rank`: ground truth of rewards and model predictions over the 50 generalized actions for three benchmarks.

definition for the best pass sequence (among the coreset) for a program is unique and consistent. We use a standard cross entropy loss to train the single-class classification model.

NVP This is the normalized value prediction method described in Section 3.2.

The last three methods (i.e., Q-value-rank, BC, and NVP) share the same **inference protocol**. Note that they all output a vector $\boldsymbol{a}$ of length 50 whose entries correspond to the pass sequences in the coreset. At inference, we roll out the pass sequences with the highest values in $\boldsymbol{a}$ one by one until our budget of 45 passes is reached. Since the pass sequences have an average length of 12.5, typically 3 or 4 pass sequences are applied (anything beyond 45 passes will be truncated). For BC and NVP, we also tried sampling pass sequences using the model output probabilities, but that resulted in worse performance. So, we simply selected sequences with the maximum values/probabilities.

4.3. Main Results

In Table 3 we present the main results of our experiments comparing our proposed method `-NVP` to various baselines. The test programs were completely held-out during both data-driven learning phases (pass sequence search and model training).

The results show that our model achieves strong performance over the prior method (`Autophase-RL-PPO`) proposed in (Haj-Ali et al., 2020b). Additionally, we can see that both the GEAN model and the normalized value prediction over the discovered coreset are needed to achieve the best performance within 45 passes. See Figure 6 in the Appendix for a visualization of the improvement in program size over the 45 passes on programs from the holdout set.

The `Oracle` shows strong performance but requires a large number of interactions with the compiler. But, this shows that the pass sequence search generalizes to new unseen programs. This is somewhat unsurprising given that the compiler's built-in hand-tuned pass list (`-Oz`) works reasonably well for most programs.

Method	#passes	$\bar{I}^{Oz}(\%)$	$\bar{I}_G^{Oz}$
Compiler (`-Oz`)	97	0	1.000
`Autophase-PPO`	45	-16.3$_{\pm 9.8}$	0.960$_{\pm 0.036}$
`GCN-PPO`	45	-10.3$_{\pm 1.0}$	0.998$_{\pm 0.003}$
`GGC-PPO`	45	-12.3$_{\pm 0.1}$	0.988$_{\pm 0.001}$
`GIN-PPO`	45	-15.1$_{\pm 5.9}$	0.972$_{\pm 0.029}$
`GAT-PPO`	45	-65.7$_{\pm 40.1}$	0.806$_{\pm 0.132}$
`GEAN-PPO`	45	-12.0$_{\pm 0.6}$	0.997$_{\pm 0.002}$
`Autophase-Q`	45	-3.9$_{\pm 0.2}$	1.006$_{\pm 0.002}$
`GEAN-Q`	45	0.7$_{\pm 1.3}$	1.016$_{\pm 0.012}$
`Autophase-BC`	45	2.9$_{\pm 0.1}$	1.045$_{\pm 0.000}$
`GEAN-BC`	45	2.8$_{\pm 0.6}$	1.045$_{\pm 0.007}$
`Autophase-NVP`	45	4.0$_{\pm 0.4}$	1.056$_{\pm 0.005}$
`GCN-NVP`	45	4.3$_{\pm 0.1}$	1.058$_{\pm 0.001}$
`GGC-NVP`	45	4.4$_{\pm 0.2}$	1.059$_{\pm 0.002}$
`GIN-NVP`	45	4.3$_{\pm 0.3}$	1.058$_{\pm 0.003}$
`GAT-NVP`	45	4.5$_{\pm 0.2}$	1.060$_{\pm 0.001}$
`GEAN-NVP`	45	4.5$_{\pm 0.1}$	1.059$_{\pm 0.000}$
Top-45	45	-7.5	0.992
Oracle	625	5.8	1.075

Table 3. Evaluation results on **held-out test set** averaged over all datasets. All methods except Compiler and Oracle baselines use 45 compiler optimization passes. `-PPO` denotes RL-PPO, and `-Q` denotes Q-value-rank. For each method, we run the training and testing 3 times with the best configuration selected by validation.

The performance of `Top-45` by itself is weak showing that in order to achieve good results in a reasonable number of passes (45) we need to leverage a general policy and search to select the most likely candidate pass sequences to evaluate.

4.4. Why Did the RL-PPO Baseline Fail?

We provide an empirical analysis of why the `RL-PPO` approaches obtain much lower performance compared to our `NVP` approaches. We *hypothesize* two possible reasons for

the failures of RL-PPO. **1) Inaccurate state-value estimation results in a high variance in training.** In the PPO algorithm, we have a policy network (the actor) to output a probability distribution over the actions. And we have a value network (the critic) for estimating the state values, where the approximation is based on regressing the cumulative reward of trajectories sampled from the current policy (Schulman et al., 2017). The update of the policy network is based on the state value outputted by the value network. Inaccurate state value estimation results in a high variance in training the policy network. Due to the stochastic nature of the value estimation that stems from the Monte Carlo sampling in cumulative reward regression, it is difficult to analyze how accurately the value network approximates the ground truth state values (which are unknown even for the programs in the training set). We alleviate this issue by analyzing the Q-value-rank approach (as introduced in Section 4.2), which can be seen as a simplified version of the value approximation in PPO. The Q-value-rank approach is simpler because the values to estimate are deterministic (i.e., the value vector $\boldsymbol{r}^p$ is fixed for a program p). Moreover, since we consider the 50 pass sequences in our coreset as 50 generalized actions, the Q-value-rank approach can be seen as the value approximation in a PPO pipeline where a trajectory consists of only a single step over the 50 generalized actions. In this sense, the Q-value-rank approach is a simplified version of the regular value estimation in PPO. Figure 3 shows that the value estimation is inaccurate for programs in the held-out test set even for Q-value-rank approach. Therefore, it is even more challenging to estimate the state values in PPO. The inaccuracy leads to a high variance in training the policy network. **2) The reward is very sparse.** As shown in Figure 2, the rewards are very sparse. Therefore, the good states (i.e., the program states with a higher chance to be optimized in code size reduction) are rarely seen by the value/policy network during rollouts. Then, the value network does not have a good value estimation for those good states, and the policy network does not converge to output a good policy for them. We conjecture these two issues are the main reason for why the RL-PPO methods obtain the worst performance as shown in Table 3.

4.5. Ablation Studies

Ablation for GEAN-NVP We perform 3 ablation experiments for GEAN-NVP, where we remove graph mixup, mask the edge embedding, and remove the type graph, respectively. The results in Table 4 show that the test MeanOverOz metric drops after removing any of the three components. Specifically, the performance drops significantly after removing the type graph, which validates its importance.

The effect of the temperature The temperature parameter T in Eq. 3 controls how sharp the target distribution is. The distribution tends to be sharper as the temperature decreases. To analyze the influence of the temperature on the generalization of the model, We vary the temperature T in training the GEAN-NVP model and report the results in Figure 4.

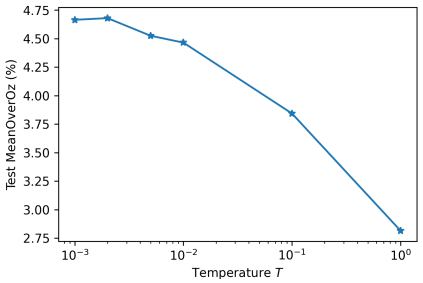

Figure 4. The effect of temperature on GEAN-NVP. Each point is obtained in a single run.

Method	Test MeanOverOz
GEAN-NVP	4.7% (0.0%)
- graph mixup	4.4% (-0.3%)
- edge embedding	4.4% (-0.3%)
- type graph	-5.3% (-10.0 %)

Table 4. Ablation on GEAN-NVP components. Each number is obtained in a single run.

5. Conclusions

In this paper, we develop a pipeline for program size reduction under limited compilation passes. We find that it is a great challenge to approximate the state values (i.e., the maximum code size reduction) for a diverse set of programs, so existing state-of-the-art methods such as proximal policy optimization (PPO) fail to obtain good performances. To tackle this problem, we propose a search algorithm that discovers a good set of pass sequences (i.e., the coreset), which generalizes well to unseen programs. Moreover, we propose to train a GNN to approximate the normalized state values of programs over the coreset, for which we propose a variant of the graph attention network, termed GEAN. Our pipeline of coreset discovery and normalized value prediction via GEAN perform significantly better than the PPO baselines.

References

Ashouri, A. H., Elhoushi, M., Hua, Y., Wang, X., Manzoor, M. A., Chan, B., and Gao, Y. Mlgoperf: An ml guided inliner to optimize performance, 2022. URL https://arxiv.org/abs/2207.08389.

Bailey, D., Harris, T., Saphir, W., Van Der Wijngaart, R., Woo, A., and Yarrow, M. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *Advances in Neural Information Processing Systems*, 31, 2018.

Brauckmann, A., Goens, A., Ertel, S., and Castrillon, J. Compiler-based graph representations for deep learning models of code. In *CC*, pp. 201–211, 2020.

Brody, S., Alon, U., and Yahav, E. How attentive are graph attention networks? In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=F72ximsx7C1.

Cereda, S., Palermo, G., Cremonesi, P., and Doni, S. A collaborative filtering approach for the automatic tuning of compiler optimisations. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 15–25, 2020.

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. End-to-end deep learning of optimization heuristics. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017.

Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., O'Boyle, M. F. P., and Leather, H. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *ICML*, 2021.

Cummins, C., Wasti, B., Guo, J., Cui, B., Ansel, J., Gomez, S., Jain, S., Liu, J., Teytaud, O., Steiner, B., et al. Compilergym: robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 92–105. IEEE, 2022.

Da Silva, A. F., Kind, B. C., de Souza Magalhães, J. W., Rocha, J. N., Guimaraes, B. C. F., and Pereira, F. M. Q. Anghabench: A suite with one million compilable c benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 378–390. IEEE, 2021.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Fursin, G. Collective tuning initiative. *arXiv preprint arXiv:1407.3487*, 2014.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pp. 3–14. IEEE, 2001.

Haj-Ali, A., Ahmed, N. K., Willke, T., Shao, Y. S., Asanovic, K., and Stoica, I. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 242–255, 2020a.

Haj-Ali, A., Huang, Q. J., Xiang, J., Moses, W., Asanovic, K., Wawrzynek, J., and Stoica, I. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 70–81, 2020b. URL https://proceedings.mlsys.org/paper/2020/file/4e732ced3463d06de0ca9a15b6153677-Paper.pdf.

Hara, Y., Tomiyama, H., Honda, S., Takada, H., and Ishii, K. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1192–1195. IEEE, 2008.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Kulkarni, S. and Cavazos, J. Mitigating the compiler optimization phase-ordering problem using machine learning. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 147–162, 2012.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

Mammadli, R., Jannesari, A., and Wolf, F. Static neural compiler optimization via deep reinforcement learning. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pp. 1–11. IEEE, 2020.

Mammadli, R., Selakovic, M., Wolf, F., and Pradel, M. Learning to make compiler optimizations more effective. In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pp. 9–20, 2021.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533, 2015.

Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. An analysis of approximations for maximizing submodular set functions—i. *Mathematical programming*, 14(1):265–294, 1978.

Planck Collaboration et al. Planck 2015 results - xiii. cosmological parameters. *Astronomy & Astrophysics*, 594:A13, 2016. doi: 10.1051/0004-6361/201525830. URL https://doi.org/10.1051/0004-6361/201525830.

Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. In Touretzky, D. (ed.), *Advances in Neural Information Processing Systems*, volume 1. Morgan-Kaufmann, 1988. URL https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., and Li, D. Mlgo: a machine learning guided compiler optimizations framework, 2021. URL https://arxiv.org/abs/2101.04808.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Wang, Z. and O'Boyle, M. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

Ward, J. and Živnỳ, S. Maximizing k-submodular functions and beyond. *ACM Transactions on Algorithms (TALG)*, 12(4):1–26, 2016.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Zhou, Y., Roy, S., Abdolrashidi, A., Wong, D., Ma, P., Xu, Q., Liu, H., Phothilimtha, P., Wang, S., Goldie, A., et al. Transferable graph optimizers for ml compilers. *Advances in Neural Information Processing Systems*, 33: 13844–13855, 2020.

A. Proofs

Lemma A.1. *The objective $J(S)$ defined in Eqn. 1*

$$\max_{|S|\leq K} J(S) = \sum_{i=1}^{N} \max_{j\in S} r_{ij} \tag{7}$$

(with the additional definition $J(\emptyset) = 0$), is a nonnegative and monotone submodular function.

Proof. Since $r_{ij} > 0$, it is clear that $J(S) \geq 0$ is nonnegative.

To incorporate the special case $J(\emptyset) = 0$, note that $J(S)$ can be written as

$$\max_{|S|\leq K} J(S) = \sum_{i=1}^{N} \max\left(\max_{j\in S} r_{ij}, 0\right). \tag{8}$$

Let $\hat{r}_{ij} = r_{ij}$ and $\hat{r}_{i,0} = 0$, then in order to prove $J(S)$ is monotone and submodular, by additivity, we only need to prove $J_i(S) := \max_{j\in S\cup\{0\}} \hat{r}_{ij}$ is monotone and submodular.

For any $A \subseteq B$, it is clear that

$$J_i(A) = \max_{j\in A\cup\{0\}} \hat{r}_{ij} \leq \max_{j\in B\cup\{0\}} \hat{r}_{ij} = J_i(B) \tag{9}$$

So $J_i(S)$ is monotone.

To prove submodularity, for any $A \subseteq B$, we comare the quatity of $J_i(A \cup \{j\}) - J_i(A)$ and $J_i(B \cup \{j\}) - J_i(B)$ for $j \notin B$.

Case 1: $\hat{r}_{ij}$ is a maximum over the subset B. In this case, then $\hat{r}_{ij}$ is also a maximum over the subset A. Then $J_i(A \cup \{j\}) = J_i(B \cup \{j\}) = \hat{r}_{ij}$, since $J_i(A) \leq J_i(B)$, we have:

$$J_i(A \cup \{j\}) - J_i(A) \geq J_i(B \cup \{j\}) - J_i(B) \tag{10}$$

Case 2: $\hat{r}_{ij}$ is a maximum over A but not in B. Then $J_i(A \cup \{j\}) - J_i(A) \geq 0$, but $J_i(B \cup \{j\}) - J_i(B) = 0$. So Eqn. 10 still holds.

Case 3: $\hat{r}_{ij}$ is neither a maximum in A or in B. Then both $J_i(A \cup \{j\}) - J_i(A) = 0$ and $J_i(B \cup \{j\}) - J_i(B) = 0$. So Eqn. 10 still holds.

By definition of submodularity (Eqn. 10), we know $J_i(S)$ is submodular and so does $J(S)$. □

B. GEAN Encoding

Our Graph Edge Attention Network (GEAN) has the following key features.

Attention with edge features We modify the attention mechanism in GAT to output an edge embedding and two node features for neighborhood aggregation. For clarity, we show a table containing the notations used in the GNN in Table 5. The feature update process can be mathematically defined by the following equations, where $M_i, i = 1, \ldots, 5$ is an encoding fully connected layer.

$$X_{ij}'^{(t+1)} = M_1(X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)}), \tag{11}$$
$$a_{ij}'^{(t+1)} = M_2(X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)}), \tag{12}$$
$$X_{ji}^{(t+1)} = M_3(X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)}), \tag{13}$$
$$a_{ji}^{(t+1)} = M_4(X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)}), \tag{14}$$
$$E_{i\to j}^{(t+1)} = M_5(X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)}), \tag{15}$$

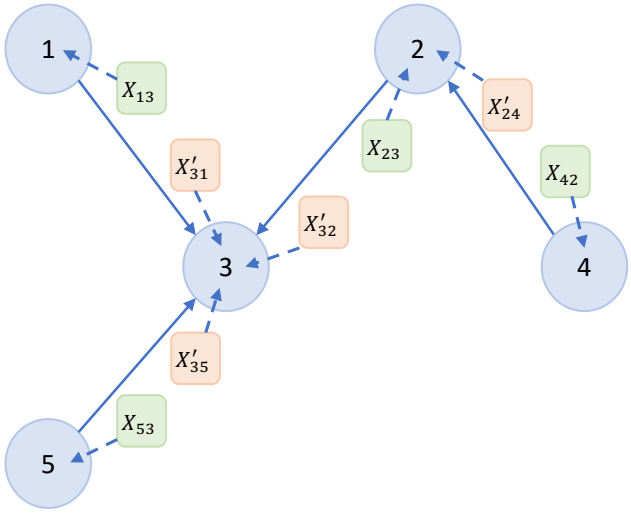

Figure 5. Graph attention. Circles denote nodes and solid arrows denote edges. Squares are the calculated features, and dash arrows represent feature aggregation. The orange/green squares denote the features to be aggregated in the target/source nodes of the edges. The edge embedding and the attention are not shown.

Notation	Meaning
$\mathcal{E}$	The set of edges in the graph
(i, j)	Edge from node i pointing to node j
$X_i^{(t)}$	Representation of node i at layer t
$E_{i\to j}^{(t)}$	Representation of the edge (i, j) at layer t
$X_{ij}^{(t)}$	Representation for node i associated with edge (i, j)
$X_{ij}^{\prime(t)}$	Representation for node i associated with edge (j, i)
$a_{ij}^{(t)}$	Raw attention associated with representation $X_{ij}^{(t)}$
$a_{ij}^{\prime(t)}$	Raw attention associated with representation $X_{ij}^{\prime(t)}$
$\alpha_{ij}^{(t)}$	Normalized attention associated with $a_{ij}^{(t)}$
$\alpha_{ij}^{\prime(t)}$	Normalized attention associated with $a_{ij}^{\prime(t)}$
$\mathcal{T}_i$	Target neighbors of node i: $\{j\|(i, j) \in \mathcal{E}\}$
$\mathcal{S}_i$	Source neighbors of node i: $\{j\|(j, i) \in \mathcal{E}\}$

Table 5. The notations in GEAN.

In words, the node-edge-node triplet, $(X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)})$, associated with edge (i, j), is encoded by fully connected layers to output 5 features, including $X_{ij}^{\prime(t+1)}$ and $a_{ij}^{\prime(t+1)}$ (a representation and attention to be aggregated in node i), and $X_{ji}^{(t+1)}$ and $a_{ji}^{(t+1)}$ (a representation and attention to be aggregated in node j), and the updated edge representation $E_{i\to j}^{(t+1)}$. Note that the features to be aggregated to a target node are marked with the $'$, and those to a source node are without the $'$ (see Figure 5). After the feature encoding, we perform an attention-weighted neighborhood aggregation for each node, which can be mathematically described by the following equations.

$$\left[\left[\alpha_{ij}^{(t+1)}\right]_{j\in\mathcal{T}_i} \middle\| \left[\alpha_{ij}^{\prime(t+1)}\right]_{j\in\mathcal{S}_i}\right] = \text{Softmax}\left[\left[a_{ij}^{(t+1)}\right]_{j\in\mathcal{T}_i} \middle\| \left[a_{ij}^{\prime(t+1)}\right]_{j\in\mathcal{S}_i}\right] \tag{16}$$

$$X_i^{(t+1)} = \sum_{j\in\mathcal{T}_i} \alpha_{ij}^{(t+1)} X_{ij}^{(t+1)} + \sum_{j\in\mathcal{S}_i} \alpha_{ij}^{\prime(t+1)} X_{ij}^{\prime(t+1)} \tag{17}$$

where $\|$ denotes concatenation.

In comparison, GAT only outputs an attention score by encoding the node-edge-node triplet: $a_{ij}^{(t+1)} = (X_i^{(t)}, E_{i\to j}^{(t)}, X_j^{(t)})$, and the feature for neighborhood aggregation is only conditioned on the neighbors: $X_{ij}^{(t+1)} = \mathrm{MLP}(X_j^{(t)})$. To summarize, our encoding approach can ensure that the GNN model is aware of the direction of the edge and that the edge embedding is updated in each layer, which helps improve the performance (as shown in Table 3).

C. Detailed Results

We report the detailed performance of some of the models in Table 6. The results are obtained from a single run with the best model configuration.

Dataset	Oracle	Top-45	Autophase-RL-PPO	Autophase-NVP	GEAN-RL-PPO	GEAN-NVP
anghabench-v1	0.7%/1.011	-1.0%/0.996	-15.9%/0.974	-0.2%/1.002	-0.8%/0.996	-0.0%/1.003
blas-v0	2.6%/1.028	-0.4%/0.997	-1.7%/0.984	2.1%/1.023	-1.0%/0.990	2.4%/1.026
cbench-v1	3.5%/1.041	-2.4%/0.984	-10.1%/0.925	-0.1%/1.008	-1.6%/0.998	2.2%/1.028
chstone-v0	9.3%/1.106	1.2%/1.016	1.3%/1.018	8.3%/1.095	5.4%/1.060	8.8%/1.101
clgen-v0	5.4%/1.060	3.1%/1.034	-0.5%/0.998	4.6%/1.051	0.3%/1.005	5.0%/1.056
csmith-v0	21.2%/1.320	-96.3%/0.851	-116.0%/0.954	21.1%/1.320	-124.6%/0.965	21.1%/1.320
github-v0	1.0%/1.011	0.2%/1.002	0.1%/1.001	0.9%/1.010	-0.2%/0.999	0.9%/1.010
linux-v0	0.6%/1.007	-0.4%/0.998	-0.5%/0.997	0.6%/1.006	-2.3%/0.989	0.6%/1.007
llvm-stress-v0	6.3%/1.087	-18.9%/0.885	-67.0%/0.731	0.7%/1.035	-17.5%/0.888	2.1%/1.045
mibench-v1	1.7%/1.020	0.0%/1.003	-2.8%/0.976	-5.8%/0.963	-0.3%/1.000	-0.1%/1.003
npb-v0	9.8%/1.159	5.7%/1.085	0.9%/1.035	5.1%/1.079	3.7%/1.068	5.5%/1.085
opencv-v0	5.2%/1.061	1.0%/1.013	0.5%/1.007	4.2%/1.051	0.3%/1.004	4.8%/1.057
poj104-v1	7.8%/1.105	3.9%/1.055	-17.5%/0.876	6.1%/1.080	-0.7%/1.008	6.3%/1.082
tensorflow-v0	6.1%/1.077	-0.2%/0.998	0.2%/1.004	5.9%/1.075	-0.2%/0.998	5.9%/1.075
Average	5.8%/1.075	-7.5%/0.992	-16.3%/0.960	3.8%/1.054	-10.0%/0.997	4.7%/1.062

Table 6. Detailed evaluation results on held-out test sets.

D. Trajectory comparison

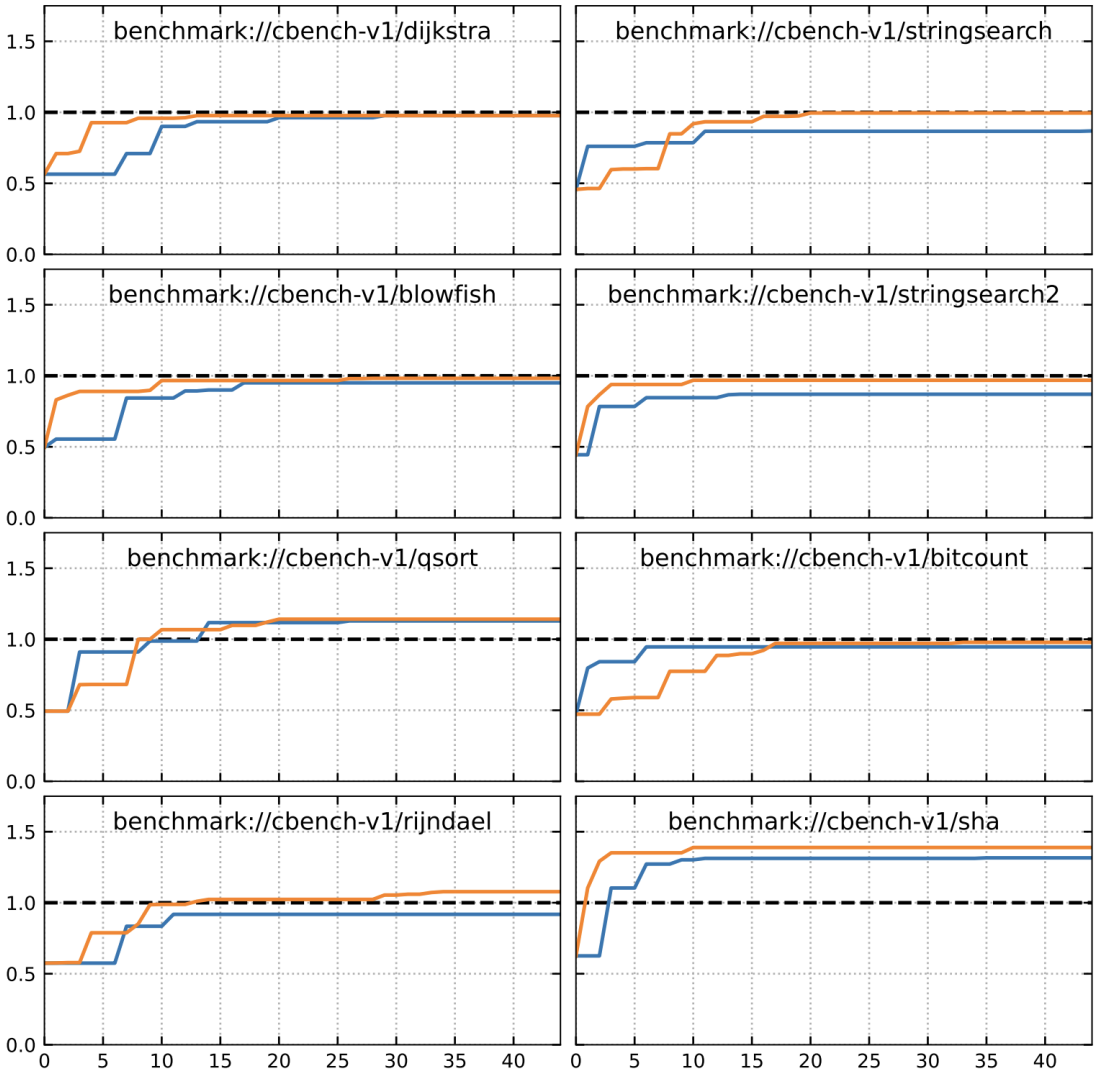

Figure 6. Program optimization example over many steps comparing the **Autophase-RL-PPO** (blue) approach with our **GEAN-NVP** (orange) approach. The dashed line represents the compiler default `-Oz` performance and higher is better. The results are obtained from a single run with the best model configuration.

E. Compiler passes

Index	Flag	Index	Flag	Index	Flag
0	-add-discriminators	42	-globalsplit	84	-lower-expect
1	-adce	43	-guard-widening	85	-lower-guard-intrinsic
2	-aggressive-instcombine	44	-hotcoldsplit	86	-lowerinvoke
3	-alignment-from-assumptions	45	-ipconstprop	87	-lower-matrix-intrinsics
4	-always-inline	46	-ipsccp	88	-lowerswitch
5	-argpromotion	47	-indvars	89	-lower-widenable-condition
6	-attributor	48	-irce	90	-memcpyopt
7	-barrier	49	-infer-address-spaces	91	-mergefunc
8	-bdce	50	-inferattrs	92	-mergeicmps
9	-break-crit-edges	51	-inject-tli-mappings	93	-mldst-motion
10	-simplifycfg	52	-instsimplify	94	-sancov
11	-callsite-splitting	53	-instcombine	95	-name-anon-globals
12	-called-value-propagation	54	-instnamer	96	-nary-reassociate
13	-canonicalize-aliases	55	-jump-threading	97	-newgvn
14	-consthoist	56	-lcssa	98	-pgo-memop-opt
15	-constmerge	57	-licm	99	-partial-inliner
16	-constprop	58	-libcalls-shrinkwrap	100	-partially-inline-libcalls
17	-coro-cleanup	59	-load-store-vectorizer	101	-post-inline-ee-instrument
18	-coro-early	60	-loop-data-prefetch	102	-functionattrs
19	-coro-elide	61	-loop-deletion	103	-mem2reg
20	-coro-split	62	-loop-distribute	104	-prune-eh
21	-correlated-propagation	63	-loop-fusion	105	-reassociate
22	-cross-dso-cfi	64	-loop-guard-widening	106	-redundant-dbg-inst-elim
23	-deadargelim	65	-loop-idiom	107	-rpo-functionattrs
24	-dce	66	-loop-instsimplify	108	-rewrite-statepoints-for-gc
25	-die	67	-loop-interchange	109	-sccp
26	-dse	68	-loop-load-elim	110	-slp-vectorizer
27	-reg2mem	69	-loop-predication	111	-sroa
28	-div-rem-pairs	70	-loop-reroll	112	-scalarizer
29	-early-cse-memssa	71	-loop-rotate	113	-separate-const-offset-from-gep
30	-early-cse	72	-loop-simplifycfg	114	-simple-loop-unswitch
31	-elim-avail-extern	73	-loop-simplify	115	-sink
32	-ee-instrument	74	-loop-sink	116	-speculative-execution
33	-flattencfg	75	-loop-reduce	117	-slsr
34	-float2int	76	-loop-unroll-and-jam	118	-strip-dead-prototypes
35	-forceattrs	77	-loop-unroll	119	-strip-debug-declare
36	-inline	78	-loop-unswitch	120	-strip-nondebug
37	-insert-gcov-profiling	79	-loop-vectorize	121	-strip
38	-gvn-hoist	80	-loop-versioning-licm	122	-tailcallelim
39	-gvn	81	-loop-versioning	123	-mergereturn
40	-globaldce	82	-loweratomic		
41	-globalopt	83	-lower-constant-intrinsics		

Table 7. A list of LLVM compiler pass indices and their corresponding command line flag, which can be obtained from the CompilerGym LLVM environment.

F. Pass sequences in the coreset

We show the 50 pass sequences in the coreset below, where the index corresponds to Table 7. The order of the sequences here is the same as the order in which they were added to the coreset.

(48, 112, 46, 110, 97, 53, 10)
(10, 53, 122, 31, 36, 111, 10, 97)
(39, 31, 53, 36, 47, 30, 33, 9, 10)
(41, 47, 104, 46)
(99, 111, 97, 40, 31, 47, 10, 36, 53)
(29, 72, 55, 103, 36, 122, 59, 30, 65, 53, 10)
(53, 36, 103, 47, 55, 9, 29, 10)
(111, 39, 10, 69, 90, 9, 29, 69, 10, 53)
(104, 39, 41, 97, 53, 10, 26, 78, 55)
(27, 36, 103, 24, 53, 97, 53, 38, 69, 97, 57, 10, 29)
(36, 38, 24, 64, 39, 53, 55, 9, 10, 118, 30)
(47, 53, 111, 57, 120, 10, 38, 21, 39)
(39, 38, 103, 117, 116, 97, 122, 10, 41, 59)
(72, 71, 31, 36, 97, 103, 78, 47, 97, 53, 41, 120, 10, 52, 97)
(31, 63, 29, 39, 93, 41, 74, 103, 120, 10, 55, 114, 55, 68, 57, 53, 95, 78, 97, 10)
(97, 65, 10, 111, 25, 74, 97, 53, 102, 120, 73, 55, 10, 53, 26)
(29, 55, 39, 61, 27, 41, 36, 25, 103, 10)
(27, 39, 64, 55, 53, 38, 122, 31, 111, 64, 10, 39, 21, 105, 36)
(53, 97, 97, 21, 65, 105, 54, 120, 10, 122, 30, 28, 39, 53)
(50, 21, 120, 97, 39, 67, 10, 29, 47, 53, 79, 36, 97, 10)
(65, 9, 55, 27, 105, 57, 103, 38, 120, 8, 29, 53, 116, 55, 39, 10, 63, 97)
(57, 9, 26, 102, 39, 8, 111, 55, 10, 104, 1)
(111, 57, 55, 120, 54, 36, 53, 122, 105, 95, 76, 47, 39, 97, 10)
(29, 103, 102, 30, 36, 61, 29, 41, 71, 10, 61, 41, 52)
(102, 10, 111, 30, 36, 121, 54, 55, 46, 50, 65, 75, 57, 9, 10, 104, 97, 53)
(56, 38, 27, 29, 50, 80, 83, 97, 55, 111, 96, 10)
(10, 64, 31, 10, 52, 111, 116, 36, 40, 48, 54, 30, 53, 114, 29, 120, 10)
(91, 115, 46, 2)
(47, 53, 36, 117, 9, 55, 74, 111, 116, 120, 9, 77, 29, 97, 10)
(27, 104, 55, 57, 26, 103, 10, 29, 31, 36, 120, 102, 53)
(102, 103, 31, 117, 59, 8, 36, 39, 75, 53, 76, 97, 70, 41, 122, 55)
(102, 53, 97, 10, 57, 71, 41, 111, 39, 71, 45, 118, 23, 53)
(30, 48, 29, 120, 103, 96, 47, 29, 78, 21, 122, 41, 36, 10)
(21, 121, 97, 38, 31, 52, 70, 53, 71, 97, 56, 111, 40, 39, 65, 10, 53)
(103, 57, 39, 53, 79, 47, 54, 97, 50, 116, 56, 53, 36, 10)
(71, 29, 111, 102, 53, 120, 38, 47, 21, 10, 120, 39, 23, 71, 40, 52)
(38, 10, 71, 39, 54, 102, 57, 103, 53, 46, 54, 116, 29, 10, 114, 41, 66)
(59, 30, 120, 79, 38, 53, 115, 10)
(99, 41, 31, 122, 36, 120, 29, 21, 111, 117, 48, 30, 10, 53)
(105, 9, 27, 55, 46, 53, 103, 76, 46, 71, 39, 41, 39, 10, 109, 30)
(59, 9, 10, 121, 114, 110, 120, 97, 10, 1, 21, 47, 53, 10, 96, 97)
(39, 99, 66, 111, 23, 25, 45, 10, 53, 75, 102, 74, 40, 105, 52, 71, 30)
(38, 47, 50, 24, 57, 30, 41, 72, 53, 56, 122, 97, 70, 15, 10, 26, 29, 53)
(53, 111, 120, 64, 36, 15, 122, 96, 121, 39, 10)
(46, 23, 120, 91)
(103, 66, 117, 47, 54, 30, 120, 36, 65, 53, 29, 96, 61, 10)
(53, 115, 86, 122, 67, 54, 30, 61, 46, 36, 10, 53)
(45, 48, 23, 91, 41, 54, 2)
(53, 91, 67, 86, 52, 61, 41, 29, 54, 10)
(123, 54, 75, 59, 10, 53, 97, 86, 80, 115, 41, 50, 10)