Deep Program Structure Modeling Through Multi-Relational Graph-based Learning

***Abstract*—Deep learning is emerging as a promising technique for building predictive models to support code-related tasks like performance optimization and code vulnerability detection. One of the critical aspects of building a successful predictive model is having the right representation to characterize the program for the given task. Existing approaches in the area typically treat the program structure as a sequential sequence but fail to capitalize on the rich semantics of data and control flow information, for which graphs are a proven representation structure.**

This paper presents POEM[1], a novel framework that automatically learns useful code representations from graph-based program structures. At the core of POEM is a new graph neural network (GNN), which is specially designed for capturing the syntax and semantic information from the program abstract syntax tree and the control and data flow graph. As a departure from all recent GNN-based code modeling techniques, our network simultaneously learns over multiple relations of a program graph. This capability enables the learning framework to distinguish and reason about the diverse code relationships, be it a data or a control flow or any other relationships that may be important for the downstream processing task.

We apply POEM to four representative tasks that require a strong ability to reason about the program structure: heterogeneous device mapping, parallel thread coarsening, loop vectorization and code vulnerability detection. We evaluate POEM on programs written in OpenCL, C, Java and Swift, and compare it against nine learning-based methods. Experimental results show that POEM consistently outperforms all competing methods across evaluation settings.

I. INTRODUCTION

Over the last two decades, machine learning has emerged as a viable means for constructing heuristics for various program-related tasks including code optimization [1]. There is now ample evidence showing that machine-learned heuristics can outperform hand-tuned approaches [2].

A key challenge for applying machine learning to programs is that it requires programs to be represented as a sequence of numerical values (such as the number and type of instructions) that serve as inputs to a machine learning model. Traditionally, such program representations were determined by experts through trials and errors. However, since programs are syntactically unbounded graph structures and that there is an infinite number of these potential features, finding the right features is a non-trivial task.

More recent studies have leveraged the advances in deep learning (DL) to model and reason about code structures [3], [4], [5], [6], [7], [8], [9]. Compared to classical machine learning approaches, DL has the advantage of not requiring expert involvement to manually tune representations for program structures; instead, it automatically captures and determines them from training samples [10].

[1]POEM = Dee**p** **C**o**de** **M**odeling.

Existing DL-based approaches for program modeling typically utilize recurrent neural networks (RNN), like the long short-term memory (LSTM) or a variant of it, to model code structures [4], [9]. Such approaches work by treating source code and its structure – for example, the abstract syntax tree (AST) – as a sequence of tokens. However, LSTM is designed for processing a sequential sequence [11] and is ill-suited for capturing the program control and data flows – which should be better represented as a graph instead of a sequence of tokens. As a result, prior methods only capture the shallow, textual structure of the source code and fail to capitalize on the rich and well-defined semantics of the program structure. To better model the complex data and program structures – which were traditionally represented as graph structures in compilers for code analysis – we need an approach that could directly operate on and learn from the graph representation of the code. Doing so will allow the learning framework to preserve and reason about much of the control and data flow information that is essential for many program-related tasks.

The first effort in this direction is the recent work presented in [12], which employs a vanilla graph neural network (GNN) to learn representations from the graph representation of the AST or the control-data flow graphs (CDFGs). This is achieved by propagating information along the graph edges defined in a graph adjacency matrix. While there may exist multiple code relationships (edges) among any given node pair, their approach only captures the graph connectivity, leaving the graph edges as untyped. As such, it cannot tell if a direct connection between two nodes is a control or a data flow, neither distinguish other relationships like order for non-commutative operations. Intuitively, such information would be essential for characterizing the program behavior for many code-related tasks. By ignoring the different relationships, their GNN approach gives marginal improvement or even worse performance compared to the LSTM alternatives [12].

We present POEM, a better approach for modeling code structures. POEM operates on graph representations of the program with the capability to learn and aggregate multiple code relationships. It is designed to maintain sequential information like token order and operand values when trading sequential representation for graph representations. POEM automatically extracts such information from the AST and the CDFGs. It then combines and abstracts the extracted information to generate a numerical feature vector that captures much of the essential information of the syntax and semantics of the target program. We use the generated embeddings as an input to a standard neural network to support downstream processing tasks like code optimization and vulnerability detection.

As a departure from prior work [12], [13], [14], POEM uses graphs to represent *both* the syntactic and semantic information of programs and employes graph-based learning methods to learn to reason over *multiple* graph structures. With POEM, syntax information is encoded from the AST and IR nodes. To maintain much of the sequential syntactic and semantic information, we augmented the AST with additional edges. These edges allow us to encode sequential syntactic relationships (e.g., "token before/after") and semantic relationships (e.g., "variable last used here", "this statement is guarded by an if condition"). In addition to the AST, we also record the control and data flow information from the CDFG. Intuitively, information collected from the source code is language-dependent but agnostic to compiler implementations. By contrast, data collected from the IR captures much of the lower-level, language-agnostic but compiler-specific information that could not be directly obtained from the source code. By combining such information, we improve the generalization ability of the learning framework, as different tasks may require knowledge at different levels.

Unlike [12] that it only uses an adjacency matrix to encode the node connectivity of the AST or CDFG, we encode different node relationships, e.g., whether it is a child-parent connection on the AST or a data flow edge in the CDFG, in different matrices and relation graphs. At the core of POEM is a novel graph neural network that can learn over multiple relationships (or edge types) simultaneously. By representing the input program as multiple relation graphs with explicit control and data flows or syntactic information, POEM captures a greater range of intra-program relations than prior graph representations. A key advantage of POEM is that it uses a learnable function to aggregate information for individual relation graphs. As the aggregation function is tuned for each relation graph, it can capture each specific code relationship more precisely. This richer set of relationships improves the model's ability in learning useful program representation, which in turns leads to better performance of downstream processing tasks. We show that POEM is highly effective in learning abstracted code representations, allowing us to solve various tasks with performance better than state of the arts.

We demonstrate the benefits of POEM by applying it to four representative tasks that require a strong ability to reason about the program structure at different levels: heterogeneous device mapping, GPU thread coarsening, loop vectorization and code vulnerability detection. We evaluate POEM on benchmarks written in OpenCL, C, Java and Swift. We compare POEM against a wide range of machine-learning techniques. Experimental results show that POEM consistently outperforms competing methods across tasks and programming languages, by giving stronger performance improvement and demonstrating a better generalization ability across evaluation tasks.

This paper makes the following contributions:

- It presents the first graph-based learning framework[2] that can simultaneously model multiple edge types of the program graph (Section II);
- It demonstrates how to use *multiple* graphs to represent both the syntactic and semantic structures of programs to support graph-based deep learning (Section II-C);
- It is the first work showing how a general graph-based learning framework can be developed to deliver consistently better performance over LSTM-based alternatives across a range of code-related tasks (Section IV).

[2]Code and data are available at: [*url redacted for double-blind review*.]

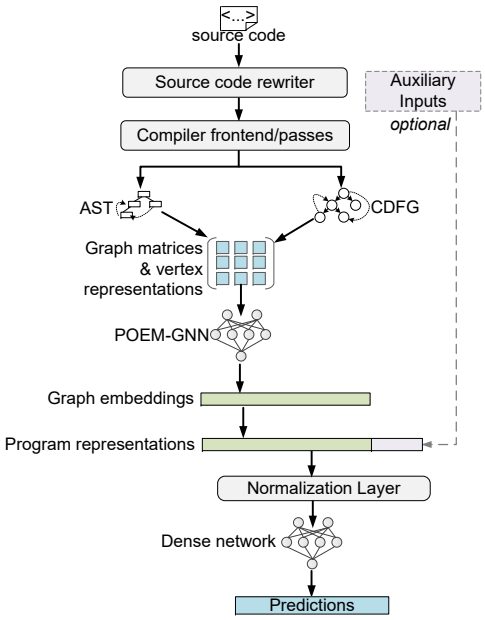


Figure 1. POEM operates on the program graph matrices and vertex representations derived from the AST and CDFG. It uses the POEM-GNN to extract useful program representations (i.e., graph embeddings), which are encoded as vectors of numerical values. The embedding vectors and the *optional* auxiliary input are concentrated and normalized and passed to decision network for a given prediction task.

II. OUR APPROACH

POEM is designed to directly operate on graph representations of the AST extracted from the source code, and the CDFG obtained from the compiler IR. As we will show later in the paper, POEM can adapt to different programming languages, hardware platforms and tasks.

A. Overview of POEM

Figure 1 depicts the architecture of POEM. It takes as input the program source code. It first uses a source rewriter extended from [4] to normalize the identifiers. Next, it builds the AST and CDFG using standard compiler passes. We extend the standard AST with additional edges to carry data and control flow information at the source code level (Section II-C). The AST and CDFG are presented as directed multigraphs, where statements, identifiers, and immediate values are vertices, and a direct relationship (e.g., parent-child, data or control flow, etc.) between two vertices is recorded as an edge. As there may exist multiple relationships (or edges) among a pair of vertices, we use a relation graph to record a specific type of relationships. In this work, we wish to capture 10 relationships from the AST and CDFG (Section II-C), leading

Table I
CODE RELATIONSHIPS CAPTURED BY POEM

Source	Relationships
AST	ASTChild, NextToken, ComputedFrom, GuardedBy, Jump, LastUse, LastLexicalUse
IR	Sequential-IR flow, data flow, and control flow

to 10 relation graphs. The vertex connectivity of a relation graph is represented as a program graph matrix (Section II-D).

The POEM-GNN takes in the program matrices and initial vertex (or node) representations to learn program representations called *embeddings* that are represented as a vector of numerical values. Like [4], the user can also optionally supply auxiliary inputs to give additional information about runtime parameters. We concentrate the graph embeddings and ancillary data to form a fixed-length feature vector, which is first normalized and then passed to a heuristic model (based on a standard fully-connected, dense neural network) to make a prediction. POEM-GNN and the dense network are trained together so that the graph representation is tuned for the task.

Unlike [12] that is only able to model the node connectivity, we extend the GNN to model multiple edge types (e.g., control, data, jump, token sequence, etc.). This capability allows POEM to distinguish different relationships of the code, whether it is an if branch or a function call. In Section IV, we show that our approach consistently outperforms prior methods that are based on the vanilla GCN [12] or LSTM [4].

B. AST and CDFG Construction

We construct the AST from the standardized source code using a compiler front-end parser (e.g., Clang for C). We construct the CDFG from the compiler IR after applying standard compiler data-flow analysis and optimizations like dead-code elimination, constant propagation, and common subexpression elimination. An AST contains syntax nodes and syntax tokens. The former corresponds to nonterminals in the language grammar, e.g., an `if` statement (`IfStmt`) and function names; and the latter corresponds to terminals like literals and constant values. The CDFG captures semantic information that reflects standard compiler transformations. Figures 2(b) and 2(c) respectively show the LLVM IR and the extracted CDFG for the OpenCL kernel (after source code rewritten) given in Figure 2(a). To simplify the IR, we replace identifiers with its type. For example, `%4` at lines 3 and 5 of Figure 2(b) will be replaced with its data type `i32`.

C. Code Relationships

We record ten relationships from the AST and the IR, listed in Table I. These include relationships that can be directly obtained from the CDFG, like the sequential order of the IR instructions, and the control and data flow. We also augment the AST with six additional edges, described as follows.

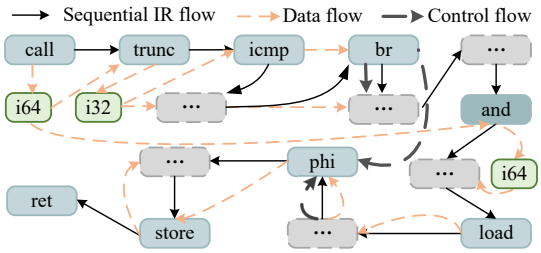
A standard AST has just one type of edges, i.e., the `ASTChild` edge that connects the children nodes with their parents. To capture additional syntax and data and control flow information of the AST, we introduce six additional

```
1  __kernel void A(__global uint4* a, __global
       uint4* b) {
2    unsigned int d = get_local_id(0);
3    if (d > 0) {
4      b[d] = a[d] + a[d + 1];
5    } else {
6      b[d] = 0;
7    }
8  }
```

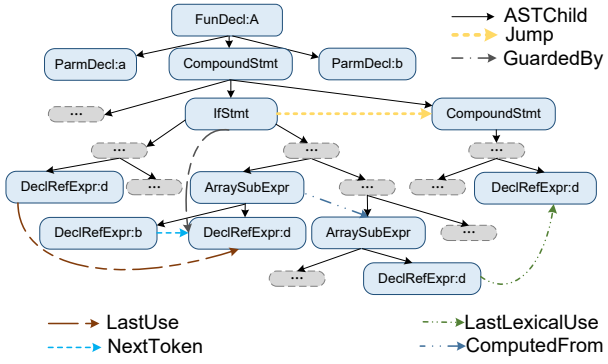
(a) An example OpenCL kernel

```
1  define void @A(i32 ,i32) {
2      %3 = tail call i64 @get_local_id(i32 0)
3      %4 = trunc i64 %3 to i32
4      %5 = icmp eq i32 %4
5      %6 = and i64 %3
6      br i1 %5, label %15, label %7
7      ...
8      %16 = phi i32 [%14, %7], [zeroinitializer
           , %2]
9      %17 = %1, i64 %6
10     store %16, %17
11     ret void
12 }
```

(b) LLVM IR



(c) Control and data flow graph (CDFG)



(d) AST with additional data and control flow edges

Figure 2. A simple, standardized OpenCL kernel (a), and the corresponding LLVM IR (b), CDFG (c) and augmented AST (d).

edges to the AST, following the method described in [15]. The additional edges are essential because they maintain local sequence order such as the ordering of variable use and operations. As the AST edges do not induce an order on children of a syntax node, we add `NextToken` edges to connect each syntax token to its successor. This edge is used to capture the order of opcode and operands for statements. For each assignment, $v = expr$, we connect v to all variable

tokens occurring in expression, $expr$, using `ComputedFrom` edges. We connect each AST token of a variable to the variable's enclosing guard expressions using a `GuardedBy` edge. For example, for the `if` statement in Figure 2(a), we add a `GuardedBy` edge from `b` to the AST node corresponding to `d > 0`. We use a `Jump` edge to connect variables that have control dependencies. The `GuardedBy` and `Jump` edges allow us to record the relation of diverging control flow. We connect all uses of the same variable using `LastUse` edges to capture the use of variables, where a special case is variables in `if` statement and we connect such type of variables using `LastLexicalUse` edges. For instance, for the `if` statement in Figure 2(a), we would link the two occurrences of `d`: one in the loop head, the other in the loop body. Figure2(d) shows the resulting AST after processing the OpenCL kernel given in Figure2(a), which is augmented with the additional edges.

Finally, for each graph edge, we also add a respective *backward* edge (by transposing the adjacency matrix), doubling the number of edges and edge types. These backward edges help with propagating information across POEM-GNN (Section II-F) and make the model more expressive.

D. Program Graph Matrices

We convert the augmented AST and CDFG to separated *relation graphs* - one graph for each of 10 relationships given in Table I. A relation graph is a directed graph, $G = (V, E)$, that contains the AST or IR node (vertices), V, and edges E, that indicates the existence of a given relationship between two vertices, such as data, control and ASTChild, etc. We use an adjacency matrix to recode the edge connections of each relation graph, 10 matrices in total. A value of 1 for matrix element $e_{i,j}$, represents there exists a direct connection or relation from node i to node j, while a value of 0 indicates the two nodes are not directly connected.

E. Vertex Representations

To capture the syntactic and semantic meanings of the relation graph vertices, we map every instructions (e.g., AST nodes like ParamDecl, IfStmt and IR opcodes), constant, and variable to a vector representation by lookup in a fixed size embedding table. To do so, we first construct a vocabulary of frequently appeared words from the training corpus, where we store the AST and IR extracted from training programs. As variable and function names and constant values can be of an arbitrary length, we encode them as tokens (i.e., letters, symbols and numbers [0-9]). During the model deployment stage, if a word of the input program is not presented in the vocabulary, it will be encoded at the token level.

Once we have constructed the vocabulary, we then apply word2vector [16] to map each word and token of the vocabulary to an embedding space of integer values. The word2vector model is trained on training benchmarks of the target programming language and compiler IR. Training is largely independent of the optimization task, whose goal is to map individual words to a point in a latent multidimensional space where words that are frequently appeared together are mapped to integer values close to each other in the space. Doing so allows us to capture much of the syntactic relation of language constructs. For example, it allows the model to learn that an `if` statement must precede an `else` statement. In this work, we map each of the tokens and words in the vocabulary to a single fixed-length embedding vector of 100 features. This vector captures many characteristics of the code, such as syntax and shadow semantic similarities.

Note that [12] uses only opcodes to compute latent representations, but ignoring information like the data types, the presence of variables and constants, all of which could be essential for many program-related tasks. POEM is designed to preserve such information by augmenting the AST and learning embeddings for these elements.

F. POEM-GNN: A Multi-Rational Graph Neural Network

The adjacency matrices and vertex embeddings of relation graphs are passed to the POEM-GNN to map the inputs to a one-dimensional embedding of 100 features. The POEM-GNN consists of several stacked GNN embedding layers based on the multi-layer perception (MLP) network, so that it can incorporate higher degree neighborhoods across relation graphs. We choose MLP because it is proven to be effective in learning embeddings for directed graphs [17], but other neural network architectures (like GRU [18]) can also be used. We use an AutoML tool [19] to automatically search for the optimal number of embedding layers from training data (see also Section IV-E1).

1) Neighborhood aggregation: Each embedding layer follows a neighborhood aggregation scheme (Figure 3b), where the d dimensional representation vector, h_v, of a graph vertex, v, is computed by recursively aggregating and transforming the representation vectors of its neighboring nodes. Vertices are initialized with the embeddings given by word2vec (Section II-E) and then exchange information by transforming their current state and sending it as a message to all neighbours in the graph. At each vertex, messages are aggregated and then used to update the associated node representation at the next embedding layer (referred to as the next iteration) [20]. The added backward edges (Section II-C) enable backwards propagation of information. After repeating this process of updating vertex states for a fixed number of iterations a *readout* function (Section II-F3) is used to aggregate the vertex representations to a single numerical vector across multiple relation graphs to be used as the program representation.

2) Multi-relation modeling: One of the novel aspects of the POEM-GNN is that it can propagate and aggregate information across multiple relation graphs, being it control and data at the IR or AST. As illustrated in Figure 3a, we achieve this by first using learnable, relation-specific MLPs, MLP_ℓ, to compute new graph states of individual relation graphs through neighborhood aggregation. Specifically, we use a feed-forward neural network to implement neighborhood aggregation for communicating the neighbor embeddings to the reference vertex (Figure 3b). We then apply a MLP-based aggregation function, MLP_{aggr}, to aggregate and update states for identical AST or IR nodes (locating using the matrix indices)

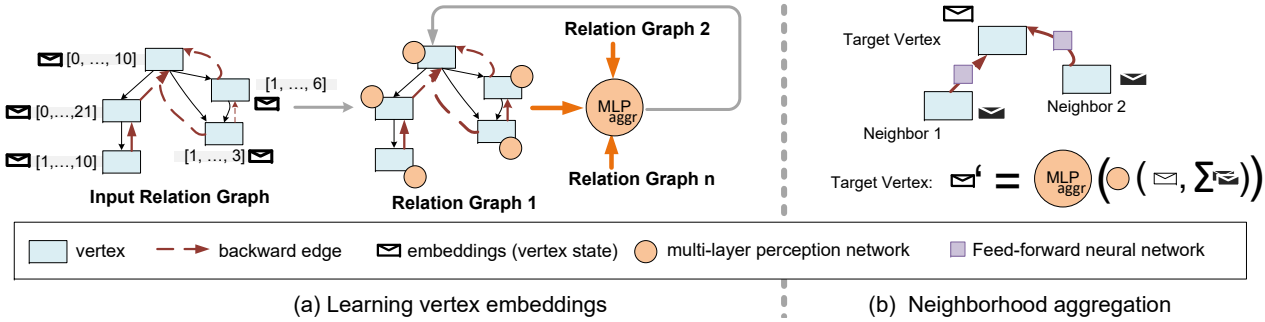


Figure 3. Using POEM-GNN to learn vertex embeddings. The initial vertex embeddings are generated using word2vec (Section II-E), which are then iteratively updated during the learning process (a) by aggregating information across neighbors and information from other relation graphs for the same vertex (b).

across relation graphs. Note that our current implementation aggregates information of AST and CDFG relation graphs as two separate groups as there is often no one-to-one mapping between AST and IR nodes after applying standard code transformations. However, the embeddings learned for the AST and CDFG will be aggregated during the readout stage.

Formally, we use forward propagation to update the state, h_v^t, for vertex, v, of a relation graph as:

$$h_v^{t+1} := \sigma(MLP_{aggr}(\sum_{\ell} \sum_{(u,v)\in A_\ell} MLP_\ell(h_u^t))) \quad (1)$$

where A_ℓ are the directed edges between a node pair, u, and v, and MLP_ℓ is a relation-graph-specific message propagation function. Note that MLP_{aggr} and MLP_ℓ are learnable functions, that are updated during the training process. The initial vertex state, h_v^0, is created using the word2vec embedding method described in Section II-E. It is worth noting that the vertex representation get more refined and global as the number of iterations increases.

3) Result readout: Once we have performed the neighbourhood aggregation procedure for a fixed number of times (determined by the number of embedding layers), we will obtain a new set of embeddings for each vertex. Through this process, the vertex knows more about their own information (features) and that of neighbouring nodes. This creates an even more accurate representation of the relation graphs. To represent the input AST and CDFG, we use a readout function to concatenate graph representations across all the neighborhood aggregation iterations and embedding layers, to form a single numerical vector, h_G, as the global program representation of all ($m = 10$) relation graphs, G_i:

$$h_G = \text{CONCAT}\left(\sum_{i=1}^{m}\left(\left\{h_{v,i}^{(t)} | v \in G_i\right\}\right) | t = 0, 1, \ldots, n\right) \quad (2)$$

where $t = 0, 1, ..., n$, is the neighborhood aggregation iterations. This readout function produces the global embedding for all relation graphs, given individual vertex embeddings.

4) Alternative modeling approaches: A naïve alternative to our approach is to apply a standard GNN to individual relation graphs and then concatenate the embedding outcomes of individual graphs. However, this approach does not allow information to be exchanged during each neighborhood aggregation iteration. In Section IV-E2, we show that this approach gives poor learning performance. As a result, simply applying the GNN presented in [12] to multiple relation graphs does address the issues of learning multiple code relations. We have also considered the recently proposed Relational Graph Convolutional Network (RGCN) [21] as it can model different relationships through multi-edge encoding. However, the RGCN has a significant drawback – the number of parameters drastically increases for larger graphs. This increase in parameters can lead to overfitting, especially when the number of training samples is limited. POEM sidesteps this problem by restricting the relation-specific learnable function to a smaller subgraph of the entire program. Furthermore, unlike the RGCN that only operates on a single graph, POEM also supports information aggregation of the distinct AST and CDFG. In Section IV-E2, we show that our approach outperforms the RGCN alternative.

G. Graph Embeddings

The graph embedding vector, produced by the POEM-GNN, together with any auxiliary data is first normalized to a range of 0 and 1 by the normalization layer. Normalization is essential as it prevents the range of single feature being a factor in its importance. The normalized feature vector is then fed to the dense network for downstream processing (Figure 1), e.g., classification. This final feature vector captures many characteristics of the code, such as semantic similarities, control and dependence flows, combinations, and analogies.

H. Train POEM

We train the POEM-GNN and the dense network together using back-propagation. Training is performed on batched training samples where each sample contains a ground-truth label. We use the standard cross-entropy loss as the objective function. This function is shown to be a good fit for sigmoid and softmax activation functions (both are also standard functions for classification) [22] used by POEM. It is defined as:

$$\mathcal{L}_G = -\sum_{i=1}^{N} y_{o,c} log(p_0, c) \quad (3)$$

where N is the number of classes (i.e., running the code on the CPU or GPU), y takes a binary value (0 or 1) that indicates if label c is the correct classification for training sample o, and p is the predicted probability for sample o to be of class c.

Table II
MACHINE LEARNING METHODS USED IN EVALUATION

Approach	Code representation	Model	Use cases
Grewe et al.[23]	Manual features	Decision Tree	Case study 1
DeepTune [4]	Source code token sequence	LSTM	Case studies 1-3
Inst2vec [14]	LLVM IR tokens	LSTM	Case studies 1-2
GNN-AST [12]	AST	Vanilla GNN	Case studies 1-3
GNN-CDFG [12]	CDFG	Vanilla GNN	Case studies 1-3
Magni *et al.* [24]	Manual features	Neural Networks	Case study 2
NeuroVectorizer [9]	Token sequence	LSTM + Reinforcement learning	Case study 3
uVuldeepecker [8]	AST	Bidirectional-LSTM	Case study 4
Lin *et al.* [25]	AST	Bidirectional-LSTM	Case study 4
POEM	AST + CDFG	multi-relational GNN	Case studies 1-4

III. EXPERIMENTAL SETUP

To demonstrate the benefit of POEM, we use it to tune performance optimization heuristics for OpenCL and C programs. To evaluate the generalization ability of POEM in modeling program structures, we also apply it to detect code vulnerabilities for C, Java and Swift. In total, we apply POEM to four case studies and compare it with nine prior machine-learning-based approaches across eight distinct hardware platforms. We use the same model structure for POEM across tasks. Table II lists the machine learning models used in the evaluation.

A. Case Study 1: Heterogeneous Mapping

The problem. This task builds a predictive model to determine if the CPU or the GPU gives faster performance for a given OpenCL kernel.

Methodology. We use the dataset of [4], which provides labeled CPU/GPU instances for 256 OpenCL kernels extracted from seven benchmark suites. The data were collected on two CPU-GPU platforms: one with an Intel Core i7-3820 CPU and and AMD Tahiti 7970 GPU, and the other has an Intel Core i7-3820 CPU and an NVIDIA GTX 970 GPU. By varying dynamic inputs, this dataset consists of 680 labeled instances on each platform. The compilation of some kernels ended with the presence of errors. We have manually fixed those broken OpenCL kernels to use the entire dataset. We apply 10-fold cross-validation train and test a model. This means we train a model on six benchmark suite and test the trained model on the remaining suite. We repeat this process ten times (folds), with each of the seven suites used exactly once as the testing data. Since the dataset in [4] is small, it may not provide sufficient training samples for a deep learning method. To evaluate on a larger training dataset, we use CLSmith, an OpenCL program synthesizer [6], to generate 12,000 valid and compilable OpenCL kernels as additional training data for deep-learning-based competing methods and POEM. This second experiment was performed on our GPU platform that uses a 3.2 GHz 6-core Xeon E5-2667 CPU and an NVIDIA Titan XP GPU. We profiled all benchmarks from the DeepTune dataset to obtain the ground-truth label on this platform.

Competitive methods. We compare POEM with five machine-learning models. These include Grewe *et al.* [23] that uses hand-tuned features, and the sequence model of DeepTune [4] that uses LSTM to extract code representations from source code token sequence. We also compare to inst2vec [14], a LSTM-based model that operates on a graph representation of the LLVM IR. For graph models, we compare to the two GNN variants presented in [12] offers GNN-CDFG and GNN-AST which operate on the CDFG and AST respectively. Like all prior work, for this case study, we use two dynamic values, the workgroup size and data size that are available to the OpenCL runtime, as the auxiliary inputs (or features) to all predictive models. Results of this case study is presented at Section IV-A.

B. Case Study 2: Thread Coarsening

The problem. This task builds a model to determine how many parallel threads should be merged together to achieve faster execution time. This is a problem known as determining the thread coarsening factor for OpenCL [24]. Here, we wish to build a model to determine for each kernel, which of the six coarsening factors, 1, 2, 4, 8, 16, and 32, should be used for a given kernel (where a factor of 1 means no coarsening).

Methodology. We replicate the setup of [24], [4], [12] by testing each approach using the labeled dataset given in [24]. This dataset contains 17 OpenCL kernels extracted from three benchmark suites. The data were collected from four distinct GPU platforms: AMD HD 5900, AMD Tahiti 7970, NVIDIA GTX 480 and NVIDIA K20c. Like [4], we use *leave-one-out* cross-validation for this task because the benchmark set is small. This works by selecting one benchmark for testing and using the remaining ones for training. This task is designed to evaluate if our approach can effectively support transfer learning [26], a technique for reusing the knowledge learned from one task to speed up the learning for another task.

Competitive methods. We compare POEM against three approaches: Magni *et al.* [24] that uses hand-tuned features, DeepTune, inst2vec, and GNN-CDFG and GNN-AST presented in [12]. To apply transfer learning, we first train an initial deep learning model on the dataset given in [4]. We then use transfer learning to fine-tune the trained model on training data used for this task. Fine-tuning is done by simply copying the learned parameters of case study one to initialize the model and then training the model as normal using cross-validation. Note that for this task, the OpenCL kernel is the sole input and coarsening factor is the predicted output. Results of this case study are presented in Section IV-B.

C. Case Study 3: Loop Vectorization

The problem. This task targets the classic compiler optimization problem of loop vectorization. It aims to build a predictive model to determine the optimal vectorization factor (VF) and the interleaving factor (IF) for individual loops. The first

Table III
DATASET FOR VULNERABILITY DETECTION.

Source	Language	#samples	#positive samples
SARD & NVD	C	156,668	78,334
	Java	60,242	30,121
GitHub	C	10,400	5,200
	Swift	4,124	2,062

parameter determines how many instructions to pack together from different loop iterations, while the latter decides the stride of the memory accesses of the packed instructions. Prior work has shown that the two parameters can have a substantial impact on the resulting vectorization performance[27], [9]. We consider 35 combinations of VF (1, 2, 4, 8, 16, 32, 64) and IF (1, 2, 4, 8, 16), which are found to be useful in [9].

Methodology. We use LLVM version 9.0 as the compiler. We configure the VF and IF on a per loop basis by placing the LLVM/Clang vectorization directives, e.g., `loop vectorize_width(VF) interleave_count(IF).` We replicate the evaluation of NeuroVectorizer [9], by using the 6,000 synthetic loops generated from the LLVM vectorization test set to train a model and then test the trained model on hand-written programs from MiBench [28] and PolyBench [29]. Our evaluation platform uses an 3.6 GHz Intel Core i7 CPU with 64GB RAM.

Competitive methods. We compare POEM against four supervised learning models: DeepTune, and the two GNN variants in [12]. We also compare to NeuroVectorizer [9], a recently proposed reinforcement-learning-based approach. NeuroVectorizer first learns the program representations through LSTM. The representations are then used by a reinforcement learner to search for the best configuration until a convergence threshold is met. Due to the nature of reinforcement learning, NeuroVectorizer can incur significant search overhead. It takes minutes to search for the vectorization configuration for a *single* loop on our evaluation platform. By contrast, our approach takes less than 100ms (including constructing the relation graphs) to make a prediction. The results are presented in Section IV-C.

D. Case Study 4: Vulnerability Detection

The problem. In this task, we build a model to detect if a given source code snippet contains one of the 2019 CWE top-25 most dangerous software errors [30] at the function level.

Methodology. As summarized in Table III, we use a dataset of 231,434 samples with source languages in C, Java and Swift, where half of the samples are vulnerable code. The vulnerable code samples are collected from the standard vulnerable code databases, including the national vulnerability database (NVD), common vulnerabilities and exposures (CVE) and open datasets collected from GitHub. The vulnerable-free samples are obtained by applying the corresponding patch to the vulnerable code. We apply 10-fold cross-validation to train and test a predictive model (see also Section III-A).

Competitive methods. For this case study, we compare POEM against two state-of-the-art deep-learning-based vulnerability detection models: uVuldeepecker [8] and Lin *et al.* [25]. Results of this case study is presented in Section IV-D.

E. Performance Report

To measure execution time for case studies 1-3, we run each test case repeatedly until the 95% confidence bound per model per input is smaller than 5%. For case study 4, we report the accuracy, and the false-positive and the false-negative rates. A false positive is when the model predicts a code snippet has a vulnerability while it does not, and a false-negative is when the model fails to detect a vulnerable code sample. For code vulnerability detection, we would like to achieve high accuracy with low false-positive and false-negative rates.

We report the geometric mean across experimental runs or test cases. The geometric mean is widely considered to be more robust and meaningful than the arithmetic mean for performance measurements [31]. Note that POEM would give better performance improvement measured by the arithmetic mean for all test cases.

F. Implementation and Training Settings

We implement POEM on Tensorflow v1.8. To build the AST, we use Clang [32] for OpenCL, C and Swift, and ANTLR [33] for Java. To extract the CDFG, we use LLVM [34] for OpenCL, C and Swift, and Soot [35] for Java.

All deep learning models were trained in an end-to-end fashion using minibatch stochastic gradient descent (SGD) and the Adam optimizer [36]. For fair comparison, we use NNI [19], an AutoML tool to determine the training hyper-parameters, including the learning rate and batch size unless these are given in the published implementation. We train the models using two NVIDIA GTX 1080 GPUs. Note that we set aside 1/10th of the training data to use for *validation* during the training process. Training terminates when the loss does not improve within 20 consecutive training epochs, or reaches a 99% accuracy on the valuation set, or meets the termination criteria given in the published implementation. For each model, we use the configuration that yields the best results on the validation set. Because we use the geometric mean instead of the arithmetic mean and as the deep learning models are initialized with random weights, performance numbers can deviate from the source publications.

IV. EXPERIMENTAL RESULTS

In this section, we first present results for the four case studies described in Section III, showing that POEM outperforms all alternative methods in each task. We then provide a detailed analysis of the working mechanism of POEM.

A. Case Study 1: Heterogeneous Mapping

Figure 4 reports the performance improvement. Results on NVIDIA GTX 970 and AMD Tahiti 7970 was obtained on the DeepTune dataset, while results on NVIDIA Titan XP were obtained by first training the models on additional synthetic

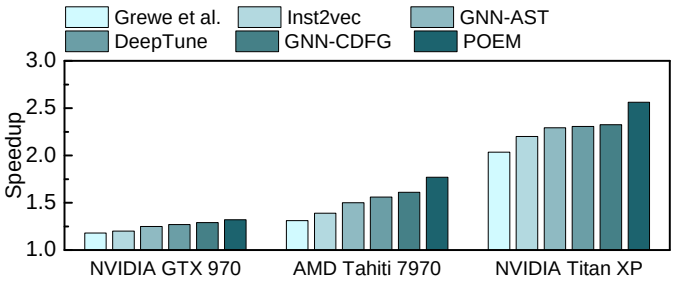


Figure 4. Speedups (geometric mean) over a GPU-only baseline for heterogeneous mapping (case study 1). POEM outperforms alternative methods on both platforms.

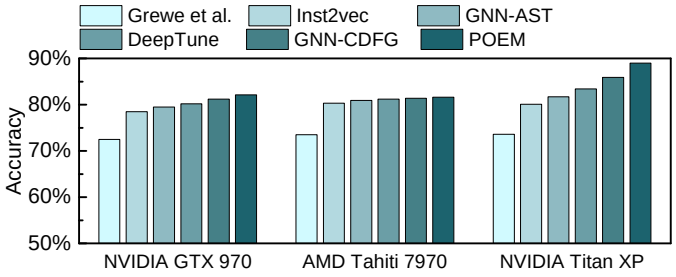


Figure 5. Prediction accuracy for heterogeneous mapping (case study 1). POEM gives the highest prediction accuracy.

OpenCL kernels and then testing the trained model on hand-written kernels from the DeepTune dataset. The baseline is a GPU-only strategy that always uses the GPU for kernel execution. As we report the geometric mean, the speedup number can deviate from the original publications.

For this case study, POEM outperforms all other approaches on all platforms. On NVIDIA GTX 970, we observe small performance improvement over the GPU-only baseline for all methods. On this platform, POEM gives the best overall speedup of 1.32, albeit its improvement is relatively small. By contrast, the benefit of using the right device on the AMD Tahiti 7970 GPU is larger. On this platform, POEM achieves a mean speedup of 1.8x, which translates to an improvement of 13% over the second-best model, GNN-CDFG. All deep-learning models benefit from additional training data on the NVIDIA Titan XP platform, where POEM delivers the highest mean speedup of 2.6x.

If we now look at the prediction accuracy given in Figure 5, we see that POEM also delivers the highest accuracy on all platforms; albeit POEM gives modest accuracy improvement on the DeepTune dataset because the number of training samples is small. However, when using a larger training dataset, it is able to boost the prediction accuracy from 82% to 89% over DeepTune and Inst2vec. We note that on the AMD platform, for most of the cases that POEM mispredicts, the difference in performance between the GPU and the CPU is small. As a result, such a misprediction has little impact on the overall performance. Overall, POEM delivers the highest mean speedup and prediction accuracy across all evaluation platforms and datasets.

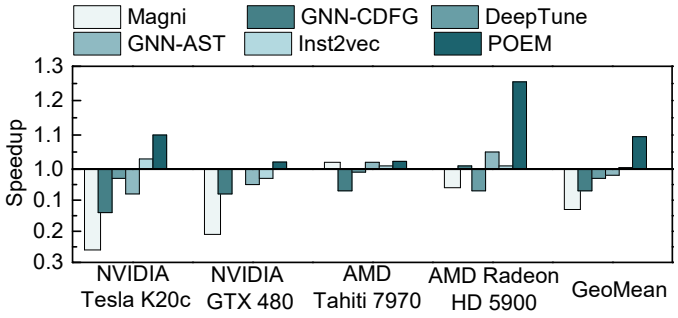


Figure 6. Performance of thread coarsening (case study 2). POEM is the only method that gives an overall speedup.

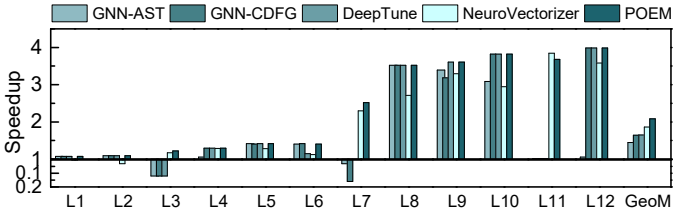


Figure 7. Speedup over the LLVM default loop vectorizer (case study 3). POEM delivers the highest overall speedup and is the best-performing model for most of the testing loops.

B. Case Study 2: Thread Coarsening

Figure 6 shows the speedup for thread coarsening over a baseline that uses a coarsen factor of 1 (i.e., no coarsening). In this task, we apply transfer learning to port the deep learning model trained for case study 1 (using the DeepTune dataset) to predicting thread coarsening . POEM is the only model that consistently delivers performance improvement across GPU platforms, albeit the improvement is modest. The modest improvement is expected as a theoretically perfect predictor would only achieve a mean speedup of 1.28x. The GNN variants deliver poor performance for this task, leading to overall slowdown on three out of four evaluation platforms. DeepTune gives marginal improvements on two GPU platforms, but its performance is far from POEM on these platforms. Notably, on Tesla K20c, POEM and inst2vec are the only predictive models that give a speedup. On HD5900, POEM gives an overall speedup of 1.26x, improving the second-best predictive model, DeepTune, by 20%. Overall, POEM achieves consistently higher speedups when compared to that of other methods. This experiment shows POEM can effectively support transfer learning when the training corpus is small.

C. Case Study 3: Loop Vectorization

Figure 7 shows the speedup for predicting loop vectorization configurations. The baseline is the LLVM default loop vectorization setting. GNN and DeepTune match or outperform NeuroVectorizer on several high-speedup test cases (L8, L9, L10), despite that NeuroVectorizer incur significantly more expensive compile-time overhead. However, GNN and DeepTune give no performance improvement or even slow down over LLVM "-O3" for several loops, including L3, L7, and L11.

Table IV
PERFORMANCE FOR CODE VULNERABILITY DETECTION (CASE STUDY 4).

Metrics		uVuldeepecker	Lin *et al.*	POEM
C	Accuracy	80.0%	88.0%	**90.9%**
	FPR	31.6%	30.5%	**3.1%**
	FNR	9.4%	**7.1%**	8.9%
Java	Accuracy	78.3%	72.0%	**84.4%**
	FPR	27.7%	45.4%	**20.7%**
	FNR	15.7%	10.3%	**8.1%**
Swift	Accuracy	77.7%	74%	**89.0%**
	FPR	21.0%	23.2%	**19.3%**
	FNR	23.6%	28.3%	**9.9%**

After having a close examination of these cases, we found that these loops contain a branch with the loop body that does not captured by DeepTune and the simply graph representation used by GNN. POEM gives or matches the best performance for all test cases, except for L11. For L11, the performance of POEM is not far from the best-performing model (i.e., NeuroVectorizer). Overall, POEM gives an average speedup of 2.08, which translates to 12% over the second best-performing model, NeuroVectorizer. Compared to NeuroVectorizer, POEM also has orders of magnitude less compile-time overhead (under a second versus 15 minutes compile time for the 12 testing loops). This indicates that the representation learned by POEM can effectively support the downstream loop vectorization task. An interesting question is if the embeddings learned by POEM can be used to improve the reinforcement search framework of NeuroVectorizer. We leave this as our future work.

D. Case Study 4: Code Vulnerability Detection

In this experiment, we apply POEM to detect vulnerabilities of function-level source code written in C, Java and Swift.

Table IV reports the higher-is-better accuracy metric and the two lower-is-better metrics: the false-positive rate (FPR) and the false-negative rate (FNR). POEM delivers the best accuracy with the lowest FPR and FNR. On the C dataset, POEM has an accuracy of over 90.9%, with a low FPR of 3.1%. POEM has a modestly higher FNR compared to Lin *et al.*, but it has a significantly lower FPR (3.1% vs 30.5%). A low PPR is important as it reduces the developer's time in investigating false alarms. For the Java and Swift datasets, all three approaches have relatively lower accuracy and higher FPR. This is largely due to the more complex language features like overriding of an external method. Such information is not captured by the initial vertex embedding method (word2vec used by the three methods). Nonetheless, POEM outperforms the other two methods across language datasets by successfully detecting more vulnerable code samples with the lowest FPR.

To illustrate the benefit of flow-sensitive representations, consider the vulnerable-free code sample given in Figure 8. Both uVuldeepecker and Lin *et al.* incorrectly classify this code snippet for containing a *double-free* vulnerability. The root cause for this *false positive* is that their sequence-based detection models have to linearize and treat the code structure as a sequential sequence of tokens, which, unfortunately, does not capture the divergence of the execution path. Consequently, they regard dynamic buffer `attr_value` (line 9) to be deallocated again after it being freed at line 6. By contrast, POEM can precisely capture the control and data flow of the target program by modeling the control and data flows. As a result, POEM correctly infers that buffer `attr_value` at line 9 is freed in a different execution path and hence will not lead to a *double-free* vulnerability.

```
1  attr_value = (char*)malloc(attr_len + 1);
2  ...
3  else if(!strcmp(attr_name, "dateadded"))
4  {
5    ae->date_added = atoi(attr_value);
6    free(attr_value);
7  }
8  else
9    free(attr_value);
```

Figure 8. Benign code sample from Github. LSTM-based models misclassify the code contains a "*double-free*" vulnerability for buffer `attr_value`.

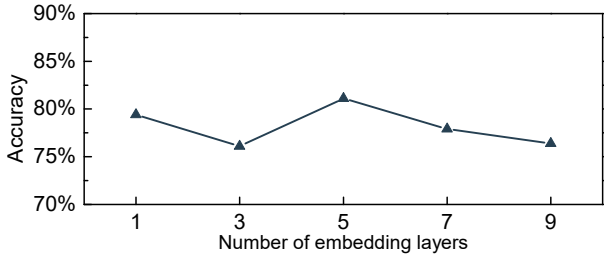


Figure 9. Performance of POEM heterogeneous mapping on AMD Tahiti 7970 as the number of embedding layers changes.

E. Model Analysis

1) Impact of embedding layers: Using heterogeneous mapping as an example, Figure 9 shows how the performance of POEM changes on the DeepTune dataset on AMD Tahiti 7970. Increasing the number of embedding layers (and hence the number of neighborhood aggregation iterations - see Section II-F) can improve the performance. However, the accuracy reaches a plateau when using five embedding layers and using more than that leads to overfitting and a drop of prediction accuracy on the validation set. The validation dataset is part of the training data but not the test dataset which is always not seen at the model design and training stage. We use the same procedure to determine the optimal number of embedding layers for each task, by comparing the resulting accuracy on the validation set. We found that using 3 to 5 embedding layers give a good performance for our tasks and using more embedding layers would require a larger training dataset.

2) Impact of implementation choices: In this experiment, we compare POEM to several variant implementations for performing heterogeneous mapping on AMD Tahiti 7970. The first is RGCN [21] that applies to the 10 code relations described in Table II-C. Unlike POEM, RGCN takes in a single adjacency matrix that encodes all the node connectivities of the AST and CDFG, and the edge is encoded using a one-hot vector for representing different relations. The second variant,

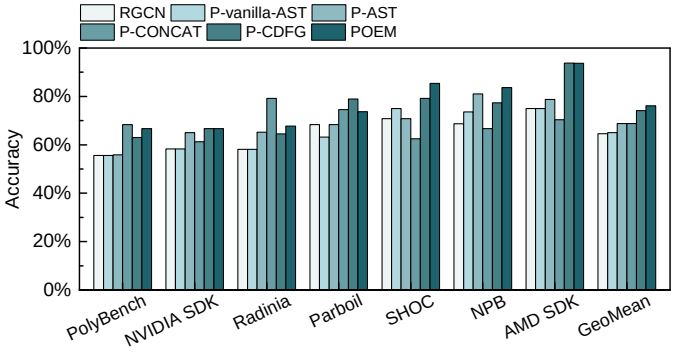


Figure 10. Comparing implementation variants of POEM on AMD Tahiti 7970 for heterogeneous mapping. Our implementation choices of POEM give the best overall performance.

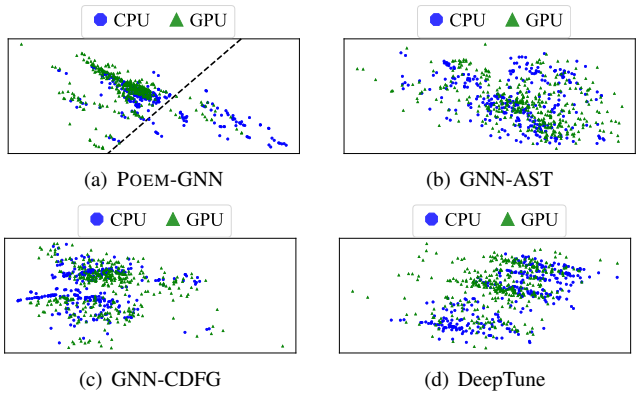


Figure 11. Visualization of the learned program representations for heterogeneous device mappings. We map the high-dimensional embedding space to a 2-dimensional space to aid clarity using t-SNE. The embeddings learned by POEM is more discriminative than the ones given by other methods, leading to a clearer boundary between the two classes.

referred to as p-vanilla-AST, operates on a standard AST (without the additional edges described in Sec. II-B). The third variant, referred to as P-AST, operates only on the augmented AST. The fourth variant, referred to as P-CDFG, operates only on the CDFG. This experiment uses heterogeneous mapping as a case study and is designed to evaluate the impact of exacting code information. The final variant, referred to as P-CONCAT, learns individual embeddings for each relation graph and then concatenates the individual embeddings for prediction. This evaluation is also known as the ablation study by the machine learning community [37].

The results are given in Figure 10. While RGCN support modelling of multiple edge relations, it is less effective for modeling a combined graph from the AST and the CDFG. Its low performance is largely due to two reasons. Firstly, a simple combination of the AST and the CDFG, which are two heterogeneous graphs, to fit the RGCN can confuse the learning algorithm. Second, RGCN requires a large number of learnable parameters and is hard to generalize. Figure 10 also shows that using the standard AST is inadequate for capturing the essential program structures. By augmenting the AST with additional control and data flow information, P-AST improves P-vanilla-AST by 4%. However, using the AST or CDFG alone is insufficient, as both give an accuracy of less than 75%. P-CDFG correctly predicts 20 kernels where P-AST fails, while it fails on other 12 kernels where P-AST succeeds. P-CONCAT also gives lower performance compared to POEM, suggesting that simply combining the embeddings of relation graphs is less effective. This experiment suggests that we need to utilize and aggregate the information of the AST and the CDFG during the learning process. POEM offers exactly such capabilities, leading the best overall performance. It also shows that our multi-relational graph learning method improves a single concatenation strategy.

3) Embedding visualization: In an attempt to examine the learned code representation qualitatively, we provide a visualization of the t-SNE-transformed feature representations [38] extracted by the POEM-GNN pre-trained on the DeepTune heterogeneous mapping dataset. The representation exhibits discernible clustering in the projected 2D space. Note that these clusters (with two different node colors) correspond to the two labels (CPU and GPU) of the dataset, verifying the model's discriminative power across different classes for this dataset. As can be seen from the diagram, the embeddings learned by POEM is more discriminative than the ones given by other methods, leading to a clearer linear boundary between the two classes (CPU and GPU).

4) Training overhead: The time for training POEM is dominated by training data collection. For example, for case study 1, it took less than 24 hours to profile over synthetic 10,000 benchmarks for labeling the data. The time in model training and hyper-parameter tuning is less than 12 hours using two modest NVIDIA 1080 GPUs on 10,000 samples. Since training is only performed once, it is a *one-off* cost.

V. DISCUSSIONS

Naturally, there is room for future work and further improvement. We discuss a few points here.

Model interpretability. Machine learning techniques, in general, have the problem of relying on black boxes. This problem is just as true for POEM. A possible approach to gain insight into why the model fails to produce the desired result is to train an interpretable model (or so called surrogate models) like linear regressor to approximate the predictions of the underlying black-box model [39].

Training samples. Deep neural networks typically require a large volume of training data to learn over. However, there is often a shortage of benchmarks. Therefore, work on benchmark synthesis [3] is orthogonal to our approach.

Training overhead. Profiling training benchmarks to generate training data could be expensive. One way of reducing the training overhead is to use active learning [40] to only profile and label training benchmarks that are likely to improve the performance of the machine-learned model.

Memory footprint. Like all GNN approaches, the memory footprint of POEM increases as the graph size increases. However, we can reduce memory pressure by using sampling methods like GraphSAGE for batched training [41], i.e., operating on a subgraph of the entire program at a time.

Other application domains. We have shown that POEM can be generalized across a range of tasks. We envision that POEM can be applied to other applications which are beyond the scope of this work. It can be applied to detect malicious code by looking for suspicious and obfuscated patterns. It can be extended to regression-based problems like predicting the potential speedup for a code transformation option. A particularly interesting research direction would be to extend POEM to model the program structure at the binary level [42] for tasks like program verification and security.

VI. RELATED WORK

Machine learning has demonstrated promises in automating the process of decision model construction for various code optimization tasks [1]. Many prior studies have shown that machine-learned models can outperform expert-crafted heuristics [43], [44], [45], [46], [47], [48], [49], [50]. However, a significant barrier still exists – programs must be represented as a set of features that serve as inputs to a machine learning tool. Traditionally, this requires experts' involvement to extract the crucial elements of the program. In some ways, we have pushed the problem from one of hand-crafting the heuristic to one of hand-crafting the right code representations.

Prior work tried to automate this process of finding representations for programs by searching useful information from the compiler IR [51], [52]. These approaches still require experts to manually define the search space of a particular compiler IR implementation. As such, they offer little help in removing human experts from the loop.

In recent year, deep learning has been employed for modeling program structures. One of the key advantages of deep learning is that it can automatically find the right feature representations from training data without human involvement [10]. Prior work for deep learning on code typically employ recurrent neural networks (RNN) like LSTM or GRU to extract program representations from token sequences. For examples, DeepTune uses LSTM to extract program representations from tokenrized OpenCL code [4] and Inst2vec applies LSTM to sequentialized IR graphs [14]. Other work uses RNNs to summarize representations from the AST [53], [54], [9] or sparse matrices [5]. While RNNs is a proven technique for natural language processing, it is mainly designed for processing a sequential sequence [11]. The challenge for treating code structures as a sequential token sequence is that statements can easily be separated by hundreds of lines of irrelevant code in sequential representations. As a result, a sequence model is unlikely to capture such long-distance dependence. As a result, RNNs are ineffective in modeling the complex program control and data flows - which should be better represented as a graph structure instead of a sequence of tokens. A graph representation not only enables the learning framework to leverage the well-defined program structures but also facilitates propagating information across the graph in a manner similar to typical compiler analyses.

An early attempt to use program graph structures for code optimization is presented in [55]. This approach requires careful hand-tuned features at the basic block level to extract information from the program graph. To predict an optimization option, it measures the similarity of the input program graph with the graphs of the training datasets. This strategy requires the training data graphs to be shipped with the compiler and hence does not scale well as the training program size increases. Furthermore, this approach does not abstract the language semantics and syntax a sufficiently high level, leading to expensive computation complexity for graph matching. Our approach eliminates the need for manual feature tuning, with a constant, lower-cost compile overhead, which is independent of the size of the training set.

Some of the most recent work has employed the recently proposed GNN to model code structures. For example, Miltiadis et al. [15] use GNN to model the AST to identify the misuse of names. The recent work presented in [12], which uses the GNN to model the AST or CFDG for OpenCL program optimization, is most closely related. Our work was conducted independently and perhaps concurrently with [12]. The approach presented in [12] uses a vanilla GNN which treat all relationships equally as a single graph edge type, whether it is a node connection, or a data or control flow. This strategy misses much of the information that could otherwise be captured. Our approach advances [12] by capturing the different relationships within a unified learning framework. Compared to [12], POEM can leverage a richer set of information by combining the AST and CFDG, leading to significantly better and more reliable performance.

In recent years, GNNs have been proposed to model and process graph structures [56], [57]. The GNN family includes a diverse class of neural network architectures based on recurrent units [58], [59] and convolutional [60] and attention [61] methods. POEM represents the first work for leveraging GNNs to learn over multiple program relationships.

VII. CONCLUSION

This paper has presented POEM, a general learning framework for supporting building machine-learned heuristics for code analysis and optimization. POEM is designed to automatically extract useful representations of programs to be used as inputs for machine learning tools. At the core of POEM is a novel Graph Neural Network (GNN) that can distinguish and aggregate information from different relationships within the program control and data flow graph and the abstract syntax tree. By providing a way to abstract and aggregate information from a well-structured program graph representation, our approach can capture a richer set of syntactic and semantic information than prior deep-learning-based approaches.

We demonstrate the generalization ability of POEM by applying it to four representative program optimization and analysis tasks spanning different programming languages. We perform extensive experiments to compare POEM with state-of-the-art approaches for each task. Experimental results show that POEM consistently outperforms prior methods, setting new state-of-the-art results for these tasks.

REFERENCES

[1] Z. Wang and M. O'Boyle, "Machine learning in compiler optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, 2018.

[2] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[3] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 86–99.

[4] ——, "End-to-end deep learning of optimization heuristics," in *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[5] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, 2018, pp. 94–108.

[6] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 200–210.

[8] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, 2019.

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

[10] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[11] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *In Proceedings of the Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[12] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020, 2020.

[13] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva, "Code mapping in heterogeneous platforms using deep learning and llvm-ir," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[14] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems*, 2018, pp. 3585–3597.

[15] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *ICLR*, 2018.

[16] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[17] Y. Yu, J. Chen, T. Gao, and M. Yu, "Dag-gnn: Dag structure learning with graph neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[18] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," in *NIPS 2014 Workshop on Deep Learning*, 2014.

[19] "Neural Network Intelligence," https://nni.readthedocs.io/.

[20] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[21] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. Van Den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European Semantic Web Conference*, 2018.

[22] Z. Zhang and M. Sabuncu, "Generalized cross entropy loss for training deep neural networks with noisy labels," in *Proceedings of the Advances in neural information processing systems*, 2018, pp. 8778–8788.

[23] D. Grewe, Z. Wang, and M. F. O'Boyle, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–10.

[24] A. Magni, C. Dubach, and M. O'Boyle, "Automatic optimization of thread-coarsening for graphics processors," in *Proceedings of the 23rd international conference on Parallel architectures and compilation techniques*, 2014, pp. 455–466.

[25] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[26] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[27] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 132–143, 2006.

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "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)*. IEEE, 2001, pp. 3–14.

[29] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes," in *InPar*, 2012.

[30] C. W. Enumeration, "2019 cwe top 25 most dangerous software errors," https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html, 2019.

[31] W. Ertel, "On the definition of speedup," in *Proceedings of the International Conference on Parallel Architectures and Languages Europe*. Springer, 1994, pp. 289–300.

[32] "Clang: A language front-end and tooling infrastructure," https://clang.llvm.org/.

[33] "ANTLR (ANother Tool for Language Recognition) ," https://www.antlr.org/.

[34] "LLVM: A collection of modular and reusable compiler and toolchain technologies," https://llvm.org/.

[35] "Soot: A framework for analyzing and transforming Java applications," http://sable.github.io/soot/.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[37] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[38] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[39] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?" explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.

[40] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Minimizing the cost of iterative compilation with active learning," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 245–256.

[41] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[42] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 363–376.

[43] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," *ACM sigplan notices*, vol. 38, no. 5, pp. 77–90, 2003.

[44] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 45–55.

[45] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 177–187.

[46] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 38–49, 2009.

[47] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[48] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. OReilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 379–390.

[49] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[50] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[51] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in *2009 International Symposium on Code Generation and Optimization*, 2009, pp. 81–91.

[52] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, 2010, pp. 197–206.

[53] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Advances in neural information processing systems*, 2018, pp. 2547–2557.

[54] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *ICLR*, 2019.

[55] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 196–206.

[56] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv*, 2018.

[57] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

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

[59] Y. Wu, X. Liu, Y. Feng, Z. Wang, R. Yan, and D. Zhao, "Relation-aware entity alignment for heterogeneous knowledge graphs," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 2019, pp. 5278–5284.

[60] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, 2016, pp. 3844–3852.

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