GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS

Chris Cummins*
School of Informatics
University of Edinburgh
c.cummins@ed.ac.uk

Zacharias V. Fisches*
Computer Science Department
ETH Zurich
zfisches@student.ethz.ch

Tal Ben-Nun
Computer Science Department
ETH Zurich
talbn@inf.ethz.ch

Torsten Hoefler
Computer Science Department
ETH Zurich
htor@inf.ethz.ch

Hugh Leather
School of Informatics
University of Edinburgh
hleather@inf.ed.ac.uk

February 28, 2020

ABSTRACT

The increasing complexity of computing systems places tremendous burden on optimizing compilers, requiring every more accurate and aggressive optimizations. Machine learning offers significant benefits for constructing optimization heuristics but there remains a gap between what state-of-the-art methods achieve and the performance of an optimal heuristic. Closing this gap requires improvements in two keys areas: a *representation* that accurately captures the semantics of a program, and a *model* architecture with sufficient expressiveness to reason over this representation.

We introduce PROGRAML — *Program Graphs for Machine Learning* — a novel graph-based program representation using a low level, language agnostic, and portable format; and machine learning models capable of performing complex reasoning over these graphs. The PROGRAML representation format is a directed multigraph in which statements and data elements are vertices, and the relations between them are edges. Distinct edge types capture whole-program control, data, and call relations, and positional embeddings differentiate branching control flow and operand ordering. PROGRAML is designed to be compiler-independent, with support currently for LLVM and XLA IRs. We employ Gated Graph Neural Network architectures to perform vertex- or graph-level classification.

We test the representational power of PROGRAML using a benchmark dataset of 250,000 LLVM IRs covering six source programming languages. We created datasets for a suite of traditional compiler analysis tasks: control flow reachability, dominator trees, data dependencies, variable liveness, and common subexpression detection. Our PROGRAML approach achieves an average F1 score across these benchmarks problems of 0.940, where a state-of-the-art LSTM-based approach achieves only 0.218. We then apply our approach to two real-world tasks — heterogeneous device mapping, and program classification — setting new state-of-the-art performance in both.

1 Introduction

The landscape of computing ecosystems is becoming increasingly complex: multi-core and many-core processors, heterogeneous systems, distributed and cloud platforms. Manually extracting performance and energy benefits from systems like these is beyond the capabilities of most programmers. In such an environment, high quality optimization heuristics are not just desirable, they are required. Despite this, good optimization heuristics are hard to come by.

*Both authors contributed equally

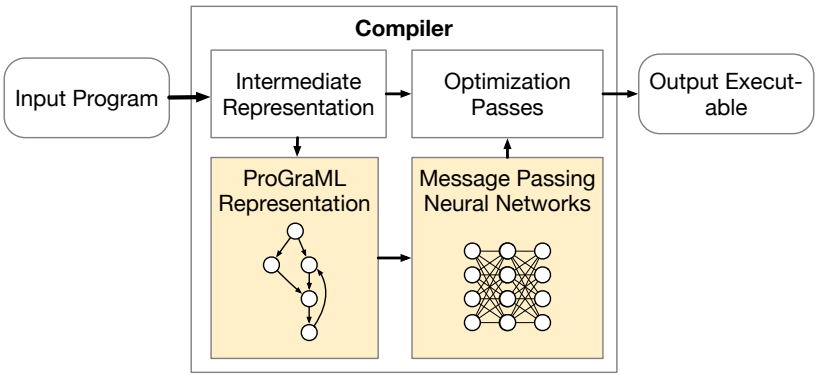


Figure 1: Our proposed approach for compiler analyses driven by graph-based deep learning. The PROGRAML representation is derived from a compiler's IR and serves as input to Message Passing Neural Networks which provide optimization decisions in place of traditional handwritten heuristics.

Designing and tuning optimization heuristics takes time, effort, and resources. To make things worse, this is a sisyphean task: even minor changes in a development toolchain might require retuning the heuristic; major changes in the software or the hardware usually require freshly designed heuristics. Machine learning offers to liberate us from this cycle by replacing fragile hand-tuned optimization heuristics with models that are inferred automatically from real performance data [1, 2]. Typically, programs are represented using a sequence of numerical features which are derived from programs using ad-hoc analysis, but such approaches fail to capture the rich semantic structure of programs, limiting the ability for models to reason over programs. Such representations prevent our automatically constructed heuristics from reproducing the reasoning of even basic compiler analyses and that in turn limits their ability to make good optimization decisions.

This is easy to see in traditional machine learned approaches where the designer explicitly chooses a program representation based only on a few properties deemed important. Recent deep learned approaches that work directly on code [3] are limited, both in the way they represent the inputs and in the way they process them. Representations based on source code and its direct artifacts (e.g. AST) put unnecessary emphasis on naming and stylistic choices that might or might not correlate with the functionality of the code. Current IR-based embeddings use compilation to remove such noise but in the process they omit important information about the program.

In both cases, the model is expected to model the flow of information in the program from representations that do not encode this information clearly. If this was not difficult enough already, processing the code representations sequentially, as all existing approaches do, makes it practically impossible. Related statements can easily be separated by hundreds of lines of irrelevant code in sequential representations. Due to *vanishing gradients* and *catastrophic forgetting*, neural networks are unlikely to even notice such very long range dependencies.

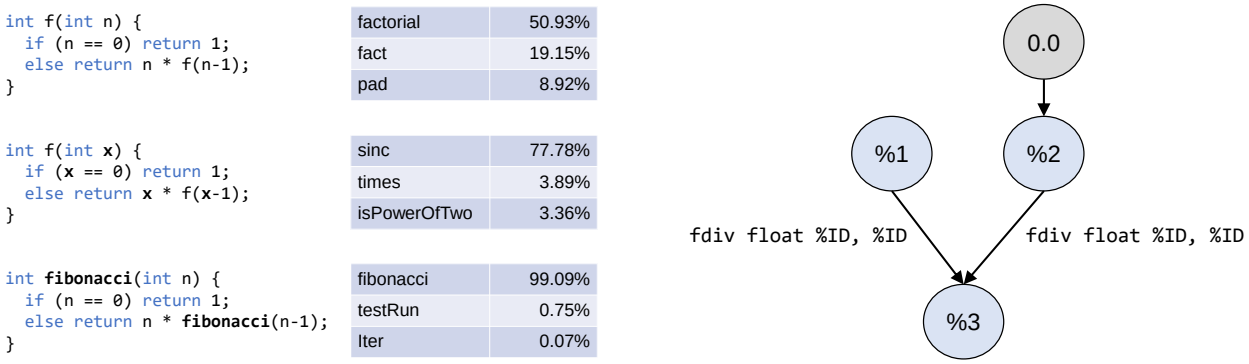
In this paper, we propose overcoming this limitation by making the program's control, data, and call dependencies a central part of the program's representation *and* a primary consideration when processing it. We achieve this by seeing the program as a graph where individual statements are connected to other statements through relational dependencies. The latent representation of each statement then is a function of not just the statement itself but the latent representations of its graph neighbors. Each statement and data element in the program is understood only in the context of the statements interacting with it. Similar techniques for learning over graphs have recently been proposed and have shown promise in other domains [4, 5, 6]. This work is the first to apply this powerful approach to compiler analyses.

With a graph-based approach, we are able to automatically learn established compiler analyses that rely on control and data flow and do it far better than existing code modeling approaches. Optimization heuristics built on top of such graph models can then natively incorporate approximate compiler analyses into its decision making, leading to superior and more powerful heuristics.

1.1 Contributions

We make the following contributions: *TODO(cec): These are out-of-date*

- We present PROGRAML, a portable graph representation of programs, and machine learning models for relational reasoning about the control, data, and call relations in programs.



(a) code2vec [10] algorithm classification.

(b) XFG [11] representation of operands.

Figure 2: Limitations in state-of-the-art learnable code representations. In (a) the model over-emphasises identifier names such that the same algorithm produces three different classifications by changing the name of a function. In (b) operand order is not captured, making the statements `x = a / b` and `x = b / a` indistinguishable. Our approach is insensitive to identifier names and preserves operand order.

- We apply our approach to a suite of established compiler analysis tasks, demonstrating 99% accuracy across programming languages. This is the first application of machine learning to compiler analyses problems.
- We demonstrate the efficacy of our approach on the challenging real-world optimization task of heterogeneous device mapping, outperforming state-of-the-art approaches by XXX.[2]

2 Motivation

There are many practical applications for machine learning over programs, from program classification to automated refactoring, and optimization. For these goals to be realised, we require machine learning models capable of reasoning about program semantics. Despite tremendous gains in recent years, state-of-the-art approaches for learning on code are not sufficiently powerful to replicate the analysis tasks that are fundamental to software. The crux of the problem lies in two central aspects of machine learning: input representation and model algorithmic complexity.

(I) Input Representation To be processed by a neural network (NN), code inputs may be encoded either directly from a source language, by way of AST analysis, or using an IR. Examples of each abound in the literature [7, 8, 9]. One state-of-the-art encoder, code2vec [10], uses AST paths to embed programs. code2vec proves highly effective at software engineering tasks such as algorithm classification, where the code was written by humans. However, as shown in Figure 2a, the trained representation can put more weight on names rather than code structure, where minute modifications completely change classification outcome. There are many benefits to such a representation, including smart pasting and correcting wrong code. However, when analysing code for optimisation, identifier names are rarely of use, whereas structure and semantics should be the primary consideration. For example, in choosing to represent code at the AST level, the code2vec representation does not capture the data or control relations between statements.

An alternate approach which emphasises semantics is Neural Code Comprehension [11], where an encoder uses Contextual Flow Graphs (XFG) built from LLVM-IR statements to create inputs for neural networks. The XFG combines partial data- and control-flow to represent the context of an individual statement. The statements are then mapped to latent-space representations using their neighbors in that graph. However, in partially combining DFGs and CFGs, the XFG representation omits important information such as order of instruction operands (making the statements `div %a, 0` and `div 0, %a` identical), and the representation fails to capture execution order, critical for many optimization tasks. In both methods of encoding programs inputs to neural networks, the representations omit information that is vital for compiler analysis, such as the control and data relations between instructions. Without this information, it is unreasonable to expect that a machine learning model would be able to reason about optimizations which this information is vital.

(II) Model Complexity The range of core operators in deep learning on code is currently confined to recurrent units (e.g. RNN, LSTM, GRU) on sequences. This poses limitations on the representation space of any such network's

[2]Code and datasets available at `https://github.com/ChrisCummins/ProGraML`

outputs. Take, for instance, dominator tree construction. An LSTM iterating forward over input code will not be able to solve the task *by definition*, as statements needs to be analysed backwards with respect to dependencies. Yet, the neural network only maintains $\mathcal{O}(1)$ memory capacity w.r.t. program length. Iterating in the opposite direction, in addition to being problem-specialised, would still not suffice, as dependencies may "vanish" in the corresponding gradients if dependent statements are far away from each other.

One way to increase the algorithmic complexity of the model is by allowing it to increase the number of code tokens that are processed simultaneously during inference. This approach, commonly used in literature by Transformer Networks [12], use preceding tokens (unidirectional encoding) or preceding and subsequent tokens (bidirectional encoding) to learn *attention matrices*. Such matrices "focus" the network on certain subsets of tokens, skipping others. However, this approach scales quadratically in memory and computation with the number of tokens.

Unlike in natural language text, the dependency structure of code is made explicit during compilation. We can thus employ domain-specific knowledge to construct the attention matrices in a scalable manner, using a graph representation of the tokens with dependencies as edges. A graph representation not only enables meaningful attention learning, but also facilitates propagating information across the graph, similarly to compiler analyses. Within the same step, a recurrent unit generates $\mathcal{O}(1)$ activations, whereas a graph NN generates $\mathcal{O}(|V|)$. To demonstrate this expressive power, control-flow reachability analysis with a sequential LSTM requires learning to memorise nodes along the way, whereas a graph NN needs only to learn an identity matrix. *TODO(Tal): Elaborate* As we shall show, this characteristic also empirically generalises better when running inference on larger programs than the network was trained on.

In this work, we extend upon the above model and representation approaches, leveraging the graph structure of IR code, taking path analysis and compilability into account.

3 Graphical Program Representation for Machine Learning

For machine learning to successfully reason over programs a suitable input representation must be provided to models. This section presents PROGRAML, a novel task- and language-agnostic graph representation that can be processed natively by machine learning models for capturing whole-program control, data, and call relations. Unlike prior approaches that rely on hand-engineered feature extractors [1, 9] or which are closely tied to the syntax of the target program language [13], our approach is both task- and language-agnostic. Our approach closely matches the representations used traditionally within compilers.

3.1 Overview

The PROGRAML representation of a compiler IR serves as the union between a program's call graph, control flow graph, and data flow graph. We represent programs as directed multigraphs with statements, identifiers, and immediate values as vertices, and relations between vertices as edges. Edges are typed to differentiate control, data, and call relations. Additionally, we augment edges with a positional embedding to encode the order of operands for statements, and to differentiate between paths in control branches.

3.2 Graph Construction

We construct a PROGRAML graph $G = (V, E)$ through $\mathcal{O}(V + E)$ traversals of a compiler IR, where V and E and the sets of vertices and edges respectively. Conceptually, the process can be divided into three stages: control flow, data flow, and call flow, though in practice the three stages can be performed in a single pass. The PROGRAML representation is designed to be compiler-agnostic and adding support for a new compiler requires only an IR parser. Currently we support IRs for LLVM [14] and XLA [15]. Figure 3 shows the graph construction approach for the LLVM-IR of a simple program for computing Fibonacci numbers.

(I) Control Flow We construct a full flow graph (Figure 3c) by inserting the statements of a program as graph vertices and edges control flow relations as edges. All control edges are numbered using an ascending sequence based on their position in the list of a vertice's outgoing control edges. For instructions with a single control successor, the position of the control edge is 0. For a branching statement with n successor statements, the control edge positions are in the range $0 \leq e_{\text{Pos}} \leq n$. We do not capture the originating function or basic block for statements as the information is captured implicitly by the control relations between statements; basic blocks are regions of statements with a single entry and exit control edge, and functions are disconnected subgraphs.

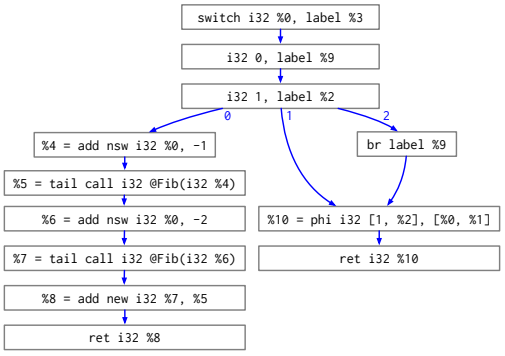
(II) Data Flow We add additional graph vertices for the immediate values and identifiers that are used by statements (Figure 3e). Data flow edges are added to capture the relation from immediate values to statements which use them

```
int Fib(int x) {
  switch (x) {
    case 0:
      return 0;
    case 1:
      return 1;
    default:
      return Fib(x - 1) +
             Fib(x - 2);
  }
}
```

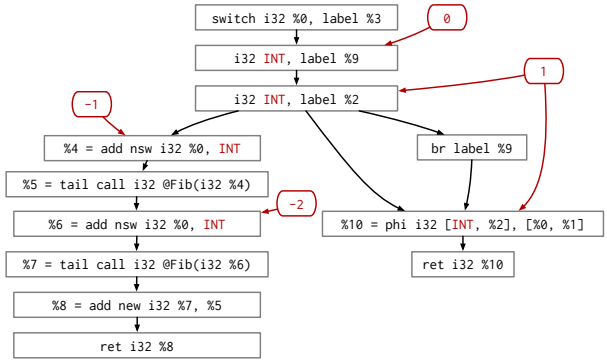
(a) Input program

```
define i32 @Fib(i32) #0 {
  switch i32 %0, label %3 [
    i32 0, label %9
    i32 1, label %2
  ]

; <label>:2:
  br label %9

; <label>:3:
  %4 = add nsw i32 %0, -1
  %5 = tail call i32 @Fib(i32 %4)
  %6 = add nsw i32 %0, -2
  %7 = tail call i32 @Fib(i32 %6)
  %8 = add nsw i32 %7, %5
  ret i32 %8

; <label>:9:
  %10 = phi i32 [ 1, %2 ], [ %0, %1 ]
  ret i32 %10
}
```

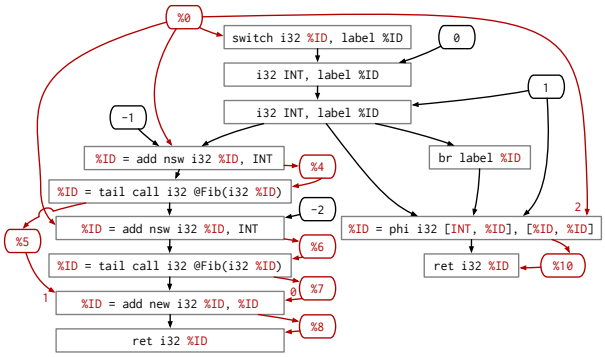
(b) Compiler intermediate representation



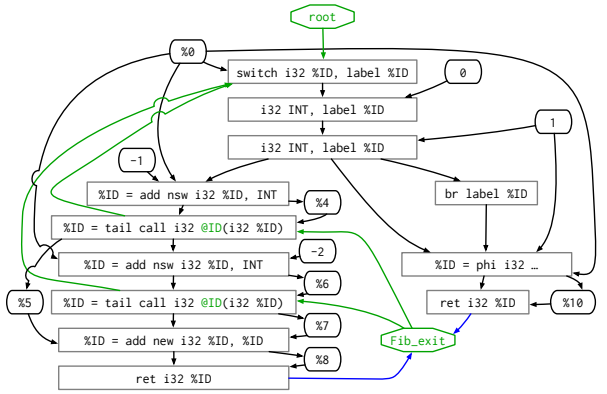
(c) Construct control flow



(d) Extract immediate values and add data flow



(e) Extract identifiers and add data flow



(f) Extract call sites and add call flow

Figure 3: *TODO(Chris): Such an ugly drawing* Construction of inter-procedural control and data flow graphs. An input program (a) is passed through the compiler front end to produce an intermediate representation (b). A full flow graph is constructed and control flow edges inserted (c). Immediate values are extracted and data flow edges inserted (d). The same process is repeated for identifiers (e). Finally, call sites are extracted and call edges are inserted from call statements to function entry points, and from function exits back to call sites (f). For functions with multiple exits, a single exit block is created and control edges added from exit statements to exit block. All control and data edges have positions, for clarity we have omitted position numbers for statements with degree one.

as operands, and from identifiers to the producer and consumer statements. Each vertex added represents a unique identifier or immediate. This way we implicitly model the scope of a variable, and unlike the tokenized representations of prior machine learning works, variables in different scopes can be differentiated. Similarly, immediate values with the same textual representation in source code (such the number 1 with `int32` and `int64` types) are represented as different vertices.

(III) Call Flow There are no control relations between functions, such that the control relations form disconnected subgraphs (the same is not true for data relations, which may cross the function boundary, for example in the case of an immediate value which is used within multiple functions of a program). Instead, the relation between a statement which calls a function and the called function is captured through call edges (Figure 3f). An outgoing call edge is inserted from the calling statement to the entry statement of a function. Return call edges are added from all terminal statements of a function to the calling statement.

For IRs with support for linking functions across translation units, such as is the case for LLVM, a statement vertex is added to the graph which represents the entry point from an external call, and connected by outgoing and returning call edges to all globally visible functions. For such IRs, a statement may call a function that is not defined in the translation unit. In such cases, a *dummy* function is created consisting of a single entry and exit statement and connected to the calling statement.

3.3 Comparison to Other Representations

The PROGRAML representation is derived from compiler intermediate representation and closely follows the representations traditionally used by compilers.

it is distinct from prior works on two keys ways:

1) first, it is language- and compiler-agnostic. Previous works have used sequential models of compiler IRs to decouple the representation from the source language, but these approaches are tightly integrated with a specific compiler

2) by using a graph structure, we more naturally capture the

Additionally, the rich representational power provided by our graph representation overcomes the limitations of prior sequential approaches. For example, by handling loops and branches in program control flow. Encoding positions for control and data edges between nodes enables statements such as `x = a / b` and `x = b / a` to be distinguished. While the loss of immediate values does not preserve full program semantics, the representational power of our graph-based approach is such that one can regenerate compilable programs from generated graphs.

TODO(Anyone): Table of representation attributes that our approach captures that state-of-art does not.

Attributes:

Control sensitive? Yes Data flow sensitive? Yes Inter-procedural? Yes Identifier sensitive? No

4 Graph-based Machine Learning

Our design mimics the *transfer functions* and *meet operators* of classical iterative dataflow analysis [16, 17], replacing the rule-based implementations with deep learning analogues which can be specialized through training to solve a diverse set of problems without human intervention or algorithm design. We formulate our system in a message passing neural network (MPNN) framework [6, 4] and implement a single unified model for all our experiments.

4.1 Overview

TODO(Zach): High level overview of the goals and what the system does, before going into the maths-y bit :-)

4.2 Model Design

The *ProGraML* model takes as input a directed graph $G = (V, E)$ with additional information as presented in Section 3 and consists of three logical phases: *(I) Input encoding*, *(II) message propagation and state updating* and finally *(III) result readout*.

(I) Input Encoding Starting from the graph representation introduced in Section 3, we capture the semantics of the nodes in the graph by mapping every normalized[3] statement or identifier node $v \in V$ to a vector representation $h_v^0 \in \mathbb{R}^{d_1}$ by lookup from a fixed size embedding table. Since the space of statements is unbounded even after normalization, our data-driven approach trades a certain amount of semantic resolution against good coverage of the vocabulary by the available datasets. In our experiments the vocabulary contains 8566 distinct token vectors. The long tail of the distribution of statements jointly maps onto a special UNKNOWN token vector in the vocabulary. Input *node-selectors*, encoded as binary one-hot vectors, are used to mark the starting point for certain analyses and are concatenated to the node embeddings. Other global input features are used as auxiliary input features at readout time in step (III), where available.

(II) Message Propagation Each iteration step is divided into a message propagation step, where each node in the graph collects learned messages m_v^t from it's undirected neighbours:

$$m_v^t = \sum_{w \in \mathcal{N}(v)} A_{\text{type}(e_{wv})} \left(h_w^{t-1} \odot \text{POS}(e_{wv}) \right) \tag{1}$$

where $\odot$ denotes the Hadamard product and $e_{wv} \in E$ is the typed edge between node w and v. In order to allow for backwards-propagation of messages, which is necessary for analyses such as finding DOMINATOR TREES, we add backward edges for each edge in the graph. For backward edges we introduce separate parameters following Li et al. [4] to enable the network to distinguish between an edge and its backward sibling. During message propagation we weight the source states h_w^{t-1} with $\text{POS}(\cdot)$, which is a constant sinusoidal position embedding [12, 18] that encodes the argument order of incoming (and outgoing) edges. This information is necessary for the network to have access to in tasks such as COMMON SUBEXPRESSION in order to distinguish non-commutative operations such as division.

Finally, the collected messages are used to update the node states in parallel according to some update function $h_v^t = U_t(h_v^{t-1}, m_v^t)$. In all our experiments, we employed a Gated Recurrent Unit (GRU) [19] with size 202 as our update function. Step (II) is iterated T times to extract node representations that are contextualized with respect to the given graph structure.

(III) Result Readout We support whole program classification, per-statement classification, and per-identifier classification by employing different *readout heads* on top of the iterated feature extraction: For graph-level classification we define a set function $R(\{h_v^T, h_v^0\}_{v \in V})$ that maps to class-scores, while for node-level inference, we separately map the extracted node features h_v^T to probabilities in parallel:

$$\begin{aligned} R_{\text{node}}(h_v^T, h_v^0) &= \sigma\left[i(h_v^T, h_v^0) \right] \cdot j(h_v^T) \\ R_{\text{graph}}(\{h_v^T, h_v^0\}_{v \in V}) &= \textstyle\sum_{v \in V} R_{\text{node}}(h_v^T, h_v^0)\,, \end{aligned}$$

where $i()$ and $j()$ are feed forward Neural Networks and $\sigma()$ is the sigmoid activation function. In the case where auxiliary graph-level features are available, those are concatenated to the readout values and fed through another feed forward Neural Network that employs Batch Normalization [20] to allow for vastly different feature scales.

4.3 Extending MPNNs to Compiler Analyses

Message Passing Neural Networks typically use a small number of propagation steps out of practical consideration for time and space efficiency [6, 4], and address problems on smaller graphs than used in this work [13]. For a large class of monotone data flow analysis problems however, it is well known that up to d$\dots d(G) + 3$ passes over the graph are required, where $d(G)$ is the *loop connectedness* of G. The *loop connectedness* captures the notion of loop-nesting depth in a program and is therefore a program-dependent, but generally unbounded quantity[4][17, 16] We address this challenge with ProGraML via a two-pronged approach: At training time, we limit the dataset to problem instances with a guarantee that the problem can be solved within a fixed number of propagation steps T[5]. Since only the first and last time step need to be preserved for inference, our system optionally iterates an arbitrary number of steps (II)at test time and we test our system on larger problem instances. Figure[6] **??** shows that ProGraML generalizes to problem instances of much larger size than it was trained on.

[3] stripping identifier names etc. this should be done in section 3 already.

[4] Given any depth-first spanning tree (DFST) of G, backwards edges are defined as those edges in G that connect a node to one of its ancestors in the DFST and $d(G)$ is the maximum number of backwards edges in any acyclic path in G.

[5] *TODO(Zach): This argument falls a little flat if we are operating on a problem where the number of needed steps is unknown, like Device Mapping... Awaiting final experiments.*

[6] *TODO(Zach): Table / Figure awaiting final experiments*

5 Experimental Methodology

We evaluate the effectiveness of our approach in two case studies. In the first, we apply our methodology to a suite of established compiler analysis tasks. These serve as demonstrations of the representational power of our approach and highlight the limitations in prior machine learning approaches. The second case study then applies the approach to the challenging real-world optimization task of heterogeneous device mapping, comparing the performance of our model against state-of-the-art machine learning-based approaches. Finally, we test our approach in a case study of classifying algorithms on the POJ-104 dataset [21].

5.1 Case Study A: Compiler Analyses

We evaluate the effectiveness of our approach in two case studies. In the first, we apply our machine learning methodology to a suite of established compiler analysis tasks. These serve as demonstrations of the representational power of our approach and highlight the limitations in prior machine learning approaches. The second case study then applies the approach to the challenging real-world optimization task of heterogeneous device mapping, comparing the performance of our model against state-of-the-art machine learning-based approaches. Finally, we test our approach in a case study of classifying algorithms on the POJ-104 dataset [21].

We construct a benchmark suite of traditional compiler analysis tasks to evaluate the representational power of our approach. We chose a diverse set of tasks to capture a mixture of both forward and backward analyses, and control-, data-, and procedure-sensitive analyses.

5.1.1 Benchmark Analyses

We selected five traditional compiler analyses to use as benchmarks for evaluating the representational power of our approach.

(I) Reachability analysis is a fundamental compiler analysis which determines the set of reachable statements given a particular starting statement. If succ(n) returns the control successors of statement n, the set of reachable statements starting at root n can be found using forward analysis:

$$\text{Reachable}(n) = \{n\} \cup \left(\bigcup_{s \in \text{succ}(n)} \text{Reachable}(s) \right)$$

(II) Dominator trees Statement n dominates statement m if every control flow path to m passes through n. A dominator tree is the set of all nodes that dominate the statment at a particular program point. Like reachability, this analysis only requires propagation of control flow, but unlike reachability, dominator trees are typically constructed using backwards analysis [22, 23]:

$$\text{Dominators}(n) = \{n\} \cup \left(\bigcap_{p \in \text{pred}(n)} \text{Dominators}(p) \right)$$

Where pred(n) which returns the control predecessors of statement n.

(III) Live-out variables A variable a is live-out of statement n if there exists some control successor of n that uses a. Given uses(n) which returns the identifier operands of n and defs(n) which returns defined identifiers, the live-out variables can be computed using:

$$\text{LiveOut}(n) = \bigcup_{s \in \text{succ}(n)} \text{uses}(s) \cup (\text{LiveOut}(s) - \text{defs}(s))$$

(IV) Data dependencies The data dependencies of statement n is the set of predecessor statements that must be evaluated to produce the operands of n. Computing data dependencies requires control and data sensitivity and is computed backwards:

$$\text{DataDep}(n) = \text{defs}(n) \cup \left(\bigcup_{p \in \text{defs}(n)} \text{DataDep}(p) \right)$$

Where $\text{defs}(n)$ returns the statements that produce operands of n.

(V) Global Common Subexpressions The identification of common subexpressions is an important analysis used for XXX. For LLVM IR we define a subexpression as a statement and its operands, ordered by either their position (for non-commutative operations), or lexicographically (for commutative operations). We thus formulate the common subexpression problem as, given a statement (which forms part of a subexpression), label any other statements which compute the same subexpression. This is an inter-procedural analysis, though identifier operands are scoped by their function.

5.1.2 Dataset

We assembled a large corpus of unoptimized LLVM IR from a variety of sources, summarized in Table 1. Our corpus comprises a range of source languages (Fortran, C, C++, OpenCL, Haskell, and Swift) and exceeds 250k files. Including code mined from GitHub [24].

We implemented each analysis or used LLVM's existing implementation to produce ground truth annotations to be used as training labels and to test the accuracy of model predictions. For every program in the corpus, we produce up to 10 unique instances by selecting different source nodes $n \in \text{Statements}(G)$ using $\lceil \min(|\boldsymbol{v}|/10, 10) \rceil$. The data set was then divided radomly into training, validation, and test programs with 3:1:1 ratio. Multiple instances derived from the same program were allocated into the same split.

5.1.3 State-of-the-art

No prior work offers the expressiveness required to perform per-statement classification, so we extended DeepTune [25], a state-of-the-art deep learning framework for whole-program classification, to enable per-statement classification. In [25] an OpenCL program is first tokenized and mapped to a sequence of embedding vectors which are then processed through a sequential LSTM model. The final state of the LSTM is optionally concatenated with program-level features features, then fed through a fully connected neural neural network to produce a program-level classification. Figure 4 shows how we extended this approach for statement-level classification of LLVM IR. We first replaced the OpenCL tokenizer using one derived from LLVM IR, resulting in a 179-element vocabulary. To adapt the approach for performing statement-level classification, we adopt a methodology proposed in XXX in which sequences of embedding vectors are grouped by their source statement. Elementwise summation is used to merge embedding vectors. The embeddings, as in [25], are trained jointly. *TODO(Anyone): Caveat of a approach is truncation of very long sequences.*

ProGraML We use the model design outlined in Section 4 for each of the compiler analysis tasks. Max number of positional edges in the corpus = 356.

5.1.4 Training Details and Parameters

All models were trained in an end-to-end fashion with the ADAM optimizer [26] with the default configuration and learning rate 0.001. The Neural Network is regularized with Dropout [27] for generalization and Batch Normalization [20] in order to be uniformly applicable to vastly different scales of auxiliary input features. Depending on the amount of available training data we report 10-fold validation splits results where noted. During training we limit the iteration depth to $T = 25$?*TODO(Zach): fill in experiments*. $d_1 = 200$? *TODO(Zach): fill in experiments*

5.2 Case Study B: Heterogeneous Device Mapping

We apply our methodology to the challenging domain of heterogeneous device mapping (Devmap). We chose this problem as there are prior machine learning approaches to compare against, using both hand-engineered features [29] and sequential models [25, 11].

We compare ProGraML against four approaches: First, with a static baseline that predicts the main mode of the dataset distribution. Second, with DeepTune [25], which is a sequential LSTM model at the OpenCL source level. Third, to isolate the impact of transitioning from OpenCL source to LLVM IR, we evaluate against a new DeepTune$_{\text{IR}}$ model,

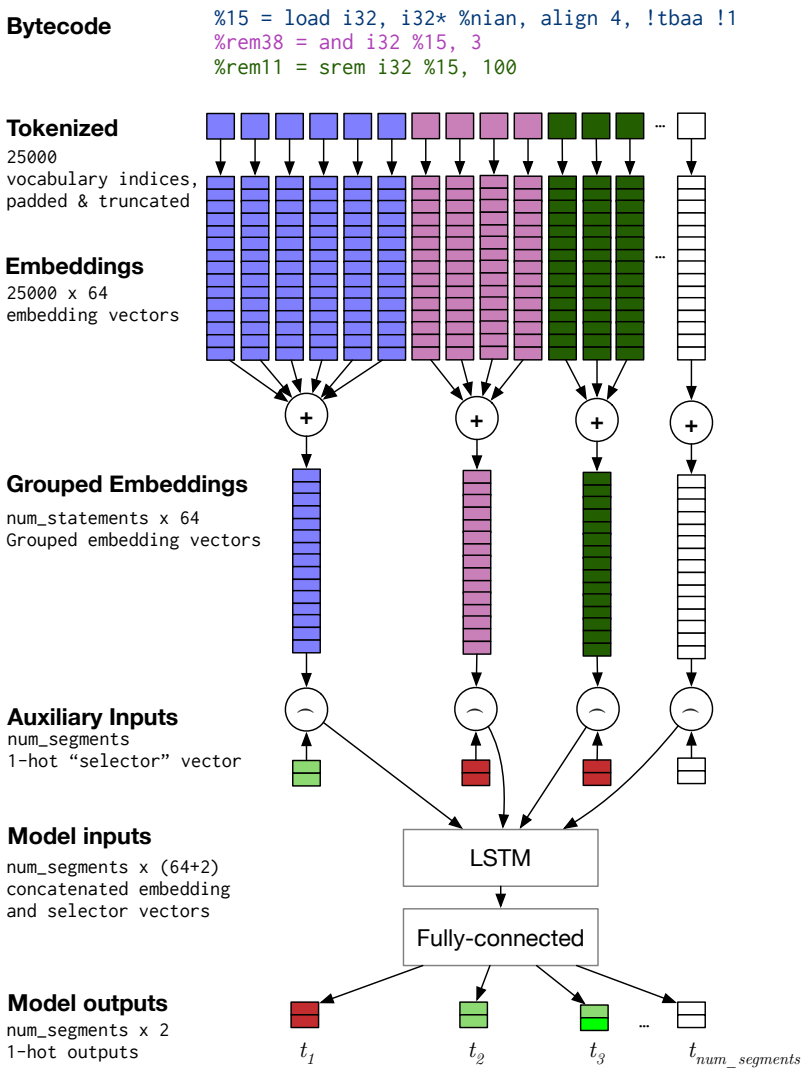


Figure 4: Extending DeepTune [25] to perform per-statement classification of LLVM IR. Whereas in prior work classification of programs was capable using the latent representation of the entire token sequence, we enable classification of arbitrary token groupings so that we can perform statement-level classification.

which is adapts DeepTune to using tokenized sequences of LLVM IR as input instead of OpenCL tokens. For this baseline, we applied the same methodology of Cummins et al. [25] to generate a tokenization of the LLVM IR in the same fashion that was used for OpenCL. Finally, we compare against the state-of-the-art approach NCC [11], which replaces the OpenCL tokenizer with a sequence of 200-dimensional embeddings, pre-trained on a large corpus of LLVM IR using a skip-gram model.

TODO [3]. OpenCL devmap with LLVM IR tokenization. The paper highlights the issue of truncated tokens. They also did dead code mutation, but only for OpenCL. For LLVM IR they just ignored the dead code. They use TF-IDF weight analysis to drop the vocabulary size in order to boost classification accuracy.

5.3 Case Study C: Algorithm Classification

In this case study, we apply our approach to the POJ-104 [21] dataset, which contains programs belonging to 104 distinct algorithms. We compare with recently published tree-based convolutional neural networks (TBCNN) [21] and NCC [11], which uses the same approach as in case study B (see above) on this dataset. To further test the expressive power of the graph-based representation against the tree-based (TBCNN) and sequential (NCC) prior work, we present additional experiments: To better understand the qualitative aspects of replacing a graph-based representation

	Source language	Domain	IR files	IR lines	Avg. nodes	Avg. edges
BLAS 3.8.0	Fortran	Scientific Computing	300	345,613	1,664	3,276
Linux 4.19	C	Operating Systems	13,851	41,332,089	1,857	3,760
OpenCL Benchmarks [25]	OpenCL	Benchmarks	256	149,779	1,027	1,970
OpenCV 3.4.0	C++	Computer Vision	400	1,168,758	3,761	7,442
POJ-104 [21]	C++	Standard Algorithms	182,815	64,518,837	312	569
Tensorflow [28]	C++	Machine learning	1,584	8,444,443	5,786	11,482
GitHub	C	Various	42,880	89,356,570	927	1,794
	Haskell		1,371	6,745,312	4,705	7,518
	OpenCL		5,188	10,472,388	2,299	5,132
	Swift		1,783	205,140	134	371
Total	—	—	**250,428**	**222,738,929**	**153,426,059**	**294,685,614**

Table 1: The ten sources of LLVM IR used to produce datasets for evaluating data flow analyses. Our corpus comprises six programming languages from functional to imperative, high-level to accelerators. The software covers a broad range of disciplines from compilers and operating systems to traditional benchmarks, machine learning systems, and unclassified code downloaded from popular open source repositories.

that captures program semantics like Contextual Flow Graphs [11] (XFG) with the more complete ProGraML representation, we adapted a GGNN [4] to directly predict algorithm classes from XFG representations of the programs. In contrast to this, Ben-Nun et al. [11] used XFG only to generate statement contexts for use in skip-gram pre-training. Here we lift this graphical representation and make it accessible to a deep neural network directly, as opposed to the structureless sequential approach in the original work (NCC). Additionally, we include a *structure-only baseline* of our ProGraML approach, where the only information on each node is whether it represents a statement or an identifier in the graph, but we refrain completely from tokenizing statements. We think that algorithm classification is a problem that lends itself especially well to judging the power of the representation *structure*, since most algorithms are well-defined independent of implementation details such as datatypes.

6 Experimental Results

6.1 Case Study A: Compiler Analysis

Data flow results in Table 2.

We report only binary precision, recall, and aggregate F1 scores as accuracy

An accuracy of 96.6% can be achieved by always predicting negative class.

LSTM is not possible for live-out variables.

Failure cases We review the failure cases here. What does this tell us about the limitations of our approach?

TODO(Chris): Plot failure rate for iterative data flow models as a function of distance from the "source" node

6.2 Case Study B: Heterogeneous Device Mapping

The performance of ProGraML and baseline models is shown in Table 3. We reused the pre-trained inst2vec-embeddings for the NCC baseline that were published with the original work, however all models themselves have been reimplemented to ensure fair comparison across different models under a unified evaluation regime and absolute performance numbers can thus deviate from the original publications.

Baseline models were trained with hyperparameters from the original works. For the ProGraML results we used 6 layers in the GNN corresponding to 6 timesteps of message propagation, while sharing parameters between even and odd layers to introduce additional regularization of the weights. We ran a sweep of basic hyperparameters which led

Problem	Analysis type	Example optimization	Model	Precision	Recall	F1
Reachability	Forwards, control flow only	Dead code elimination	DeepTune$_{IR}$	0.520	0.497	0.504
			ProGraML	**0.997**	**0.995**	**0.996**
Dominator Trees	Backwards, control flow only	Global Code Motion	DeepTune$_{IR}$	0.721	0.081	0.138
			ProGraML	**0.985**	**0.693**	**0.781**
Instruction Dependencies	Backwards, control and data flow	Instruction scheduling	DeepTune$_{IR}$	0.999	0.136	0.236
			ProGraML	**1.000**	**0.988**	**0.993**
Live-out Variables	Backwards, control and data flow	Register allocation	DeepTune$_{IR}$	—	—	—
			ProGraML	**1.000**	**0.999**	**0.999**
Global Common Subexpressions	Forwards, statement and operand values and positions	Global Common Subexpression Elimination	DeepTune$_{IR}$	**1.000**	0.123	0.214
			ProGraML	0.965	**0.925**	**0.930**
Average	—	—	DeepTune$_{IR}$	0.810	0.209	0.273
			ProGraML	**0.989**	**0.920**	**0.940**

Table 2: Benchmark compiler analyses results using two approaches: (a) DeepTune$_{IR}$, a sequential model adapted to work at the LLVM bytecode level for statement-level classification, and (b) PROGRAML our approach. Clearly, the graph representation significantly outperforms a sequential model across all problems. For the Global Common Subexpressions analysis, DeepTune$_{IR}$ achieved a higher precision than PROGRAML by predicted only the root statement as a component in a subexpression, avoiding false-positives.

	Accuracy	Precision	Recall	F1
Static Mapping	58.8%	0.35	0.59	0.44
DeepTune [25]	71.9%	0.72	0.72	0.72
DeepTune$_{IR}$	73.8%	0.76	0.74	0.75
inst2vec [11]	80.3%	0.81	0.80	0.80
ProGraML	**86.6%**	**0.89**	**0.87**	**0.88**

(a) AMD

	Accuracy	Precision	Recall	F1
Static Mapping	56.9%	0.32	0.57	0.41
DeepTune [25]	61.0%	0.69	0.61	0.65
DeepTune$_{IR}$	68.4%	0.70	0.68	0.69
inst2vec [11]	78.5%	0.79	0.79	0.79
ProGraML	**80.0%**	**0.81**	**0.80**	**0.80**

(b) NVIDIA

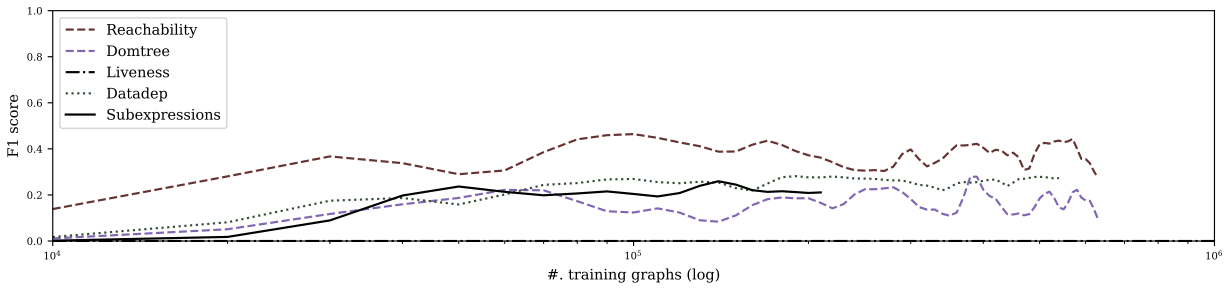
Table 3: Five approaches to predicting heterogeneous device mapping: (a) Static Mapping (b) DeepTune [25], a sequential model using tokenized OpenCL, (c) DeepTune$_{IR}$, the same model adapted for tokenized LLVM IR, (d) inst2vec, which uses pre-trained statement embeddings, and (e) PROGRAML, our approach.

us to use frozen inst2vec statement embeddings [11] and to exclude the use of position representations. Both of these hyperparameter choices help generalization by reducing the complexity of the model. This is not surprising in light of the fact that the dataset only contains 680 samples derived from ca. 200 unique kernels. ProGraML was trained with the ADAM optimizer with default parameters, a learning rate of 10^{-3} and a batch size of 18,000 nodes for 300 epochs (resulting in ca. 12000 iteration steps of the optimizer). Additionally we found dropout [30] with a rate of 0.1 on the weights of the message propagation function to be beneficial as well.

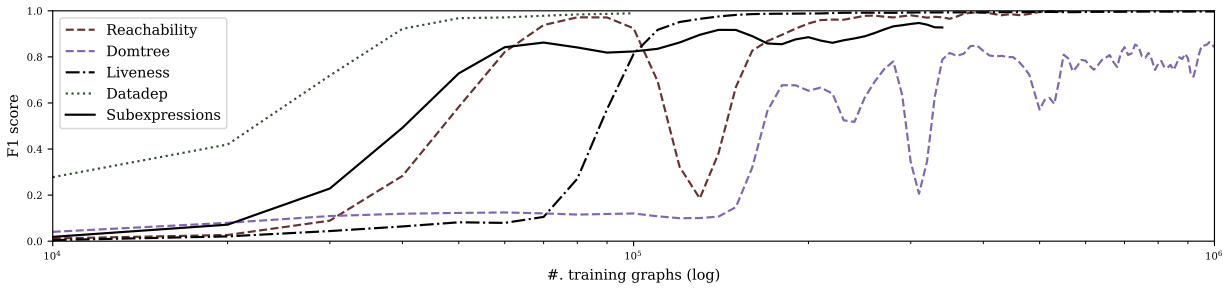
To evaluate performance of all models reported in Table 3, we used 10-fold cross-validation with rotating 80/10/10 splits by training on 80% of the data and selecting the model with the highest validation accuracy. For the ProGraML result, we repeat this for all hyperparameter configurations and picked the configuration with the best average validation performance. Performance on the unseen tenth of the data is reported.

Failure cases We review the failure cases here. What does this tell us about the limitations of our approach?

TODO(Chris): Plot failure rate for iterative data flow models as a function of distance from the "source" node



(a) DeepTune$_{\text{IR}}$



(b) ProGraML

Figure 5: The F1 score of compiler analysis models on a 20k-graph validation set, as a function of the number of training graphs from 10k to 1M. Each model was given six hours to train on machine with GTX 1080 GPU. Training terminated after six hours, if accuracy on the validation set reached 99.99%. We applied a Gaussian filter ($\sigma = 1$) to better reveal the trend of each model.

TODO(Chris): Sexy plot.

Figure 6: Confusion matrices for ProGraML models on benchmark compiler analysis tasks.

6.3 Case Study C: Algorithm Classification

7 Related Work

Graph Neural Networks Graph Neural Networks comprise a diverse class of neural networks that learn by propagating information along edges [31, 32, 33]. Approaches based on Recurrent Neural Networks (RNNs) [4, 6] as well as convolutional [34, 35] and attention-based methods [36, 37] exist. GNNs as a family have been shown to have enough expressive power to address difficult combinatorial problems like the graph isomorphism test [38] at least as well as the Weisfeiler-Lehman algorithm [39]. For a comprehensive review of GNNs, refer to the excellent review by Battaglia et al. [33]. *TODO(Chris): How does our work differ? What is new?*

Structured program representations for ML: In order to perform machine learning on programs, prior work has employed methods from Natural Language Processing and represented programs as a sequence of lexical tokens [40, 41, 25]. However, it has been observed [42, 13, 43] that it is critical to capture the structured nature of programs and syntactic (tree-based) as well as semantic (graph-based) representations have been proposed [7]. There is a line of research that considers program representations based on Abstract Syntax Trees (ASTs): Dam et al. [44] annotate nodes in the AST with type information and employ Tree-Based LSTMs [45] for program defect prediction. Both Raychev et al. [42] and Alon et al. [10, **?**] use path-based abstractions of the AST as program representations, while Allamanis et al. [13] augment ASTs with a hand-crafted set of additional typed edges and use GGNNs [4] to learn downstream tasks related to variable naming. The history of representing programs as graphs for optimization goes

TODO(Chris): Sexy plot.

Figure 7: Model error rate on analyses tasks, as a function of the distance of predicted vertex from the source vertex.

	TBCNN [21]	inst2vec [11]	XFG		ProGraML	
Metric			i2v	rnd	struct-only	master
Test Error [%]	6.0	5.17	4.56	4.29	3.87	**3.68**
Improvement [%]		0.0	11.8	17.0	25.1	**28.8**

Table 4: **Algorithm Classification Error on POJ-104 [?]):** Our two proposed models are distinct only in their embedding layers: *Graph-i2v* uses inst2vec embeddings, while *Graph* jointly learns the embeddings. The results denoted by Surface Features and TBCNN are reproduced from Mou et al. [**?**].

back to Program Dependence Graphs (PDGs) [46], which remove superfluous control flow edges to ease optimization with a compact graph representation. A more contemporary precursor of our CDFGs are ConteXtual Flow Graphs (XFGs) [11], which combine control flow with dataflow in order to learn unsupervised embeddings of LLVM IR [47] statements. While our CDFGs are designed to extend the concepts of XFGs, CDFGs preserve the notion of argument order and include nodes for immediate values as well as identifiers and all control flow edges. *TODO(Chris): XFG designed to easily understand program semantics. CDFGs, in combining CG, CFG, and DFG, are much closer to traditional compiler representations.* Another approach is taken by IR2Vec [48], which defines an LLVM IR-specific statement representation that elegantly models part-of-statements as relations. However, in order to compute the values of the embeddings, IR2Vec requires access to the type of data flow analyses that our approach is learning from data alone. Finally, another line of research considers modelling binary similarity via control flow graphs (CFGs) with an adaptation of GNNs called Graph Matching Networks [49].

8 Conclusions

As the demand for aggresively optimizing compilers grows, there is an increasing pressure on our ability to automate parts compiler construction? Machine learning promises to relieve compiler developers of the burden of hand-crafting optimization heuristics, but the success of the technique is predicated on

8.1 Future Work

Solve new problems. x86 -> graphs for optimization of legacy binaries. Other LLVM tasks: branch prediction, function inlining, loop parallelization, etc.

Increased representation power: embeddings for immediate value types and values. Separate statements from operands so that the vocabulary covers all LLVM instructions. Augment static program graphs with traces / profiling info.

Increases "solution" power: learn graph-to-graph transformations. E.g. perform global code motion.

References

[1] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A Survey on Compiler Autotuning using Machine Learning. *CSUR*, 51(5), 2018.

[2] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe. GraphIt - A High-Performance DSL for Graph Analytics. *arXiv:1805.00923*, 2018.

[3] Francesco Barchi, Gianvito Urgese, Enrico Macii, and Andrea Acquaviva. Code mapping in heterogeneous platforms using deep learning and LLVM-IR. *Proceedings - Design Automation Conference*, (June), 2019.

[4] Y. Li, R. Zemel, M. Brockscmidt, and D. Tarlow. Gated Graph Sequence Neural Networks. *arXiv:1511.05493*, 2015.

[5] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling Relational Data with Graph Convolutional Networks. In *ESWC*, 2018.

[6] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural Message Passing for Quantum Chemistry. In *ICML*. PMLR, 2017.

[7] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A Survey of Machine Learning for Big Code and Naturalness. *CSUR*, 51(4), 2018.

[8] Z. Chen and M. Monperrus. A Literature Study of Embeddings on Source Code. *arXiv:1904.03061*, 2019.

[9] Z. Wang and M. O'Boyle. Machine learning in Compiler Optimization. *Proceedings of the IEEE*, 106(23), 2018.

[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning Distributed Representations of Code. In *POPL*, 2018.

[11] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *NeurIPS*, 2018.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. In *NIPS*, 2017.

[13] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs. In *ICLR*, 2017.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. IEEE, 2004.

[15] C. Leary and T. Wang. XLA: TensorFlow, compiled. In *TensorFlow Dev Summit*, 2017.

[16] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3), 1977.

[17] K. D. Cooper, T. J. Harvey, and K. Kennedy. Iterative Data-flow Analysis, Revisited. Technical report, 2004.

[18] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional Sequence to Sequence Learning. In *ICML*. PMLR, 2017.

[19] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP*, 2014.

[20] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*. PMLR, 2015.

[21] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*, 2016.

[22] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flow Graph. *TOPLAS*, 1(1), 1979.

[23] S. Blazy, D. Demange, and D. Pichardie. Validating dominator trees for a fast, verified dominance test. In *ITP*, 2015.

[24] C. Cummins, P. Petoumenos, W. Zang, and H. Leather. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE, 2017.

[25] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end Deep Learning of Optimization Heuristics. In *PACT*. IEEE, 2017.

[26] D. P. Kingma and J. L. Ba. Adam: a Method for Stochastic Optimization. In *ICLR*, 2015.

[27] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving Neural Networks by Preventing Co-adaptation of Feature Detectors. *arXiv:1207.0580*, 2012.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-scale Machine Learning. In *OSDI*, 2016.

[29] D. Grewe, Z. Wang, and M. O'Boyle. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *CGO*. IEEE, 2013.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *JMLR*, 15, 2014.

[31] M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. In *IJCNN*. IEEE, 2005.

[32] S. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1), 2009.

[33] P. Battaglia, J. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetii, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv:1806.01261*, 2018.

[34] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*, 2016.

[35] T. N. Kipf and M. Welling. Semi-supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907*, 2017.

[36] P. Velickovic, G. Cucurull, A. Casanova, A. Romera, P. Lio, and Y. Bengio. Graph Attention Networks. In *ICLR*, 2018.

[37] G. Wang, R. Ying, J. Huang, and J. Leskovec. Improving Graph Attention Networks with Large Margin-based Constraints. *arXiv:1910.11945*, 2019.

[38] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph neural Networks? In *ICLR*, 2019.

[39] B. Weisfeiler and A. A. Lehman. A Reduction of a Graph to a Canonical Form and an Algebra Arising During this Reduction. *Nauchno-Technicheskaya Informatsia*, 2(9), 1968.

[40] M. Allamanis and C. Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *MSR*, 2013.

[41] M. Allamanis. *Learning Natural Coding Conventions*. PhD thesis, 2016.

[42] V. Raychev, M. Vechev, and A. Krause. Predicting Program Properties from "Big Code". In *POPL*, 2015.

[43] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A General Path-Based Representation for Predicting Program Properties. In *PLDI*. ACM, 2018.

[44] H. K. Dam, J. Grundy, T. Kim, and C. Kim. A Deep Tree-Based Model for Software Defect Prediction. *arXiv:1802.00921*, 2018.

[45] K. S. Tai, R. Socher, and C. D. Manning. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *arXiv:1503.00075*, 2015.

[46] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *TOPLAS*, 9(3), 1987.

[47] C. Lattner. Introduction to the LLVM Compiler System. In *ACAT*, 2008.

[48] V. Keerthy S, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Spkant. IR2Vec: A Flow Analysis based Scalable Infrastructure for Program Encodings. *arXiv:1909.06228*, 2019.

[49] L. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *ICML*. PMLR, 2019.