Compiler-Based Graph Representations for Deep Learning Models of Code

Alexander Brauckmann
TU Dresden
Dresden, Germany
alexander.brauckmann@tu-dresden.de

Andrés Goens
TU Dresden
Dresden, Germany
andres.goens@tu-dresden.de

Sebastian Ertel
Barkhausen Institut
Dresden, Germany
sebastian.ertel@barkhauseninstitut.org

Jeronimo Castrillon
TU Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

In natural language processing, novel methods in deep learning, like recurrent neural networks (RNNs) on sequences of words, have been very successful. In contrast to natural languages, programming languages usually have a well-defined structure. With this structure compilers can reason about programs, using graphs such as abstract syntax trees (ASTs) or control-data flow graphs (CDFGs). In this paper, we argue that we should use these graph structures instead of sequences for learning compiler optimization tasks. To this end, we use graph neural networks (GNNs) for learning predictive compiler tasks on two representations based on ASTs and CDFGs. Experiments show that this improves upon the state-of-the-art in the task of heterogeneous OpenCL mapping, while providing orders of magnitude faster inference times, crucial for compiler optimizations. When testing on benchmark suites not included for training, our AST-based model significantly outperforms the state-of-the-art by over 12 percentage points in terms of accuracy. It is the only one to perform clearly better than a random mapping. On the task of predicting thread coarsening factors, we show that all of the methods fail to produce an overall speedup.

CCS Concepts • **Software and its engineering → Compilers**; • **Computing methodologies → Neural networks**; Natural language processing; *Graphics processors.*

Keywords Deep Learning, Compilers, Graphs, LLVM

CC '20, February 22–23, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377894

ACM Reference Format:
Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377555.3377894

1 Introduction

The last decade has seen tremendous improvements in machine learning, especially due to deep learning methods, which do not rely on manually-specified features of the data to be learned, but are able to learn what features are important on their own. Deep learning methods have revolutionized several fields like image recognition or natural language processing. While progress has been made in compiler optimization and tasks related to programming languages, deep learning still plays a comparably modest role in these fields.

Most approaches to deep learning in compiler optimization [21] borrow ideas from the successful deep learning methods in natural language processing. This is very reasonable, as the similarities between natural languages and programming languages have fueled cross-fertilization of these two fields for many years. Prominent examples feature Long Short Term Memory (LSTM) architectures on sequences of tokens [5], or learning vector embeddings for instructions [3], in analogy to the successful *word2vec* method [16], also based on sequence models and LSTM. A comprehensive survey can be found in [1]. However, the nature of the analysis required for compiler optimization has shown to require very specific structural properties, which might be neglected by the sequential nature of these methods. For example, the polyhedral model looks into the geometric structure of the lattices of array indices to find optimal re-orderings of nested loops. This model has proven to be very successful in optimizing such nested loops [20]. More generally, compilers base most analysis on non-sequential data structures like abstract syntax trees (ASTs) and control- and dataflow graphs (CDFGs). We argue that these kinds of structures are very different

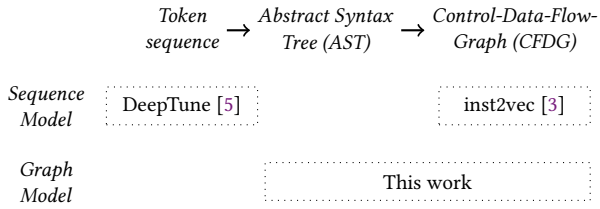

Figure 1. Representations of code in the compiler with their respective machine learning models.

from the sequences of words that compose natural language. Data dependencies can be crucial on parts of the code that are *far* apart in the code string, and details in a small part of the source, like the precise array indices in loops, can have a tremendous impact on some optimizations. For this reason, focusing on the similarities to natural language processing for deep learning in compilers might be ill-advised.

In this paper, we propose to re-evaluate the representations of code we use in deep learning for compilers. We believe that decades of research in the compiler domain have identified data structures (cf. Figure 1) that expose the aspects important for optimization better than the token sequences that are analogous to word sequences in natural languages. We thus propose to use graph-based neural networks, to explicitly capture these graph representations of code. Graph neural networks have shown to be significantly better for reasoning tasks, even when these are formulated in natural language terms [9]. We believe these reasoning tasks, like inference from a series of logical statements, structurally resemble the kind of analysis required for compiler optimization. We study two concrete graph-based architectures (Section 4), using ASTs or CDFGs as input (Section 3). We compare them with traditional methods on two complex tasks (Section 5), deciding whether OpenCL kernels should execute on a CPU or a GPU, and predicting optimal thread coarsening factors. With orders of magnitude smaller inference times, our methods clearly outperform two different state-of-the-art models ([5, 16]) in this task, with the CDFG model yielding the best accuracy overall. Additionally, we consider an alternative experimental setup to test aspects about the generality of the learned heuristics. Instead of randomly choosing training and test kernels from the available pool, we make sure these sets feature kernels from disjoint benchmark suites. With this setup, we show that our AST-based model is the only one that clearly outperforms the equivalent of a coin-toss to decide the kernel mapping when generalizing across benchmark suites.

Finally, We also test on a second task, namely predicting thread coarsening factors. With much less data and only modest opportunities for speedup, this task showcases the current limitations of deep learning in compilers. Our graph-based models do not outperform the state-of-the-art. However, none of the considered models are able to produce an overall speedup (> 1) with their predictions.

2 Related Work

Very recently, the deep learning model class of Graph Neural Networks (GNNs) has emerged, which enables the extraction of task-specific features from graph-structured data. In many tasks where information can be represented as a graph, GNN-based models have proven to be superior to sequence-based models that operate on a serialized representation of the graph [10, 13]. By applying various established architectures for sequences and euclidian data to the GNN model, recurrence- and convolutional-based variants of the GNN have been proposed and shown to support integration into complex architectures to achieve discriminative and generative tasks where they are part of an end-to-end optimization process [22].

A recent method has combined graph neural networks and code representation learning for node-level prediction tasks in the domain of software-engineering. In a variable name- and misuse prediction task based on a AST-graph representation, they have shown to outperform recurrent neural networks [2].

Probabilistic models on source code have successfully been used for solving various tasks on source code in the research area of compiler optimization [21]. While the optimizations themselves are performed with formal, algorithmic methods, probabilistic models are used for selecting and tuning them to achieve better performance [6]. Also in adjacent research areas, probabilistic models have solved tasks where statistical properties of source code are desirable, e.g. auto-completion [19] and variable naming [18]. Similarly, they have been used to provide approximations for problems that due to their enormous complexity are hardly solvable with formal methods, such as software defect prediction [14] and tuning of compiler heuristics [3, 5, 15, 17], albeit not without caveats [7].

Early models for predicting compiler optimizations rely on features that are manually designed by experts [6, 15, 17] and automatically selected from a set of candidates [11]. Those features are typically quantities of instructions in compiler intermediate representations (IRs). To extract features that capture the complex dependencies, several models relying on deep learning [12] have been proposed: Cummins et al [5] train an LSTM recurrent neural network model to predict compiler-internal optimization heuristics based on a program represented as a raw sequence of C programming language tokens. Ben-nun et al [3] adopt the notion of the distributional hypothesis, which is commonly known as word2vec [16] to produce embeddings, i.e. mappings to real vectors, on a graphical representation based on LLVM IR. This results in a closeness of the embeddings for code that shares similar contexts in the large body of open-source software projects that the embeddings have been trained on. Subsequently, the embeddings are used in more complex downstream tasks as input to an LSTM recurrent neural

network model. However, the semantic information of the compiler IR is only being used to produce the embeddings in the first phase, and not part of the optimization of the downstream task. Therefore, the embeddings are not optimized for the specific task. Figure 1 shows how these models compare conceptually to our work.

To the best of our knowledge, no machine learning model that uses the semantic information of compiler-internal analyses in an end-to-end setting has been studied.

3 Compiler-Based Representations

A compiler typically uses several representations of code at different stages in order to enable different analyses and optimizations. In this section, we will discuss three of the most common representations at the different stages and how we expect their properties to affect machine learning tasks.

Consider the following OpenCL kernel as a running example for this section:

```
__kernel void Add(__global const int* x,
                  __global const int* y,
                  __global int* z, const int d) {
 const int id = get_global_id(0);
 if (id < d)
   z[id] = x[id] + y[id];
}
```

The code for this kernel contains very obvious information, like the numbers and types of input variables for the kernel, but also very subtle information, like the indentation style of the code, which while semantically irrelevant for the compiler, can say something about the person writing the program. In this paper, we are taking the point of view of the compiler, and thus will not take into consideration such information that does not change the semantics of the program, like the indentation style, or more importantly, the identifiers in the code. Thus, we will consider code that is normalized for identifiers, using the methodology described in [5]. It standardizes all identifiers and removes whitespaces. The kernel above becomes:

```
__kernel void A(__global const int* a,
__global const int* b, __global int* c,
const int d) {const int e = get_global_id
(0); if (e < d) c[e] = a[e] + b[e];}
```

When compiling this kernel, a typical compiler[1] will transform this input string of character into multiple different representations at various stages, namely as a sequence of tokens, an abstract syntax tree, and a control and dataflow graph. We will consider these three representations and emphasize the focus for machine learning for code, as the representation is what permits the ML model to gather the relevant information. If a piece of information is not included in a representation, it cannot be learned. On the other hand, if a particular kind of information is very prominent in a representation, an ML model is more likely to learn patterns based on this information.

[1] in this work, concretely, we use LLVM and Clang in version 7.1.0

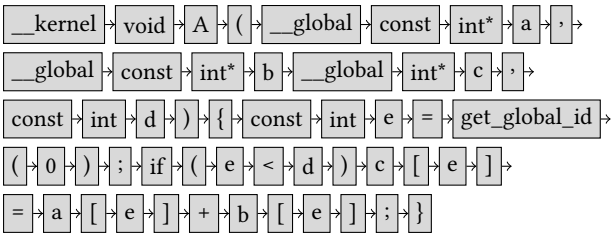

Figure 2. An example kernel as a sequence of tokens.

3.1 Sequence of Tokens

Conceptually, a compiler first turns a string of characters into a sequence of tokens. Figure 2 shows the example kernel as a sequence of tokens. Boxes are used to represent tokens in the figure, and arrows point to the next token in the sequence. This illustrates both the sequential nature of a token string, in particular as its structural similarities to strings of characters. In terms of machine learning, this sequential nature can be an advantage and a drawback at the same time. Data organized in a sequential manner has been well-studied in ML, and powerful methods exist that yield good results on sequential data. The ordering of the sequence of tokens also faithfully represents the ordering of the code as written by the programmer. This ordering contains information on the thought process of the programmer, which might be valuable for analyzing and characterizing code. However, the ordering may overconstrain the representation, enforcing an artificial order on code fragments that could otherwise execute in a different order.

3.2 Abstract Syntax-Tree

After parsing, an abstract syntax tree (AST) on its own loses some information. In particular, if an AST does not include the identifier strings, then it is impossible to tell when two identifiers refer to the same data. To deal with this, instead of embedding the identifiers, we enhance the AST with (labeled) *dataflow edges*, which connect two nodes in the AST that refer to the same data.

Figure 3 shows our example kernel as an AST. We can see how the first two arguments to the kernel (`x` and `y` in the original code) are indistinguishable in the AST. Because addition is commutative, swapping these two arguments yields the same result. In the token sequence, because tokens have to have an order, this symmetry is broken by imposing `x` as coming before `y`. The AST, on the other hand, removes this synthetic dependency: swapping the arguments is an isomorphism of the AST and our dataflow-enhanced version. This is an example of how the AST abstracts away structure that is not syntactically relevant, otherwise imposed by the

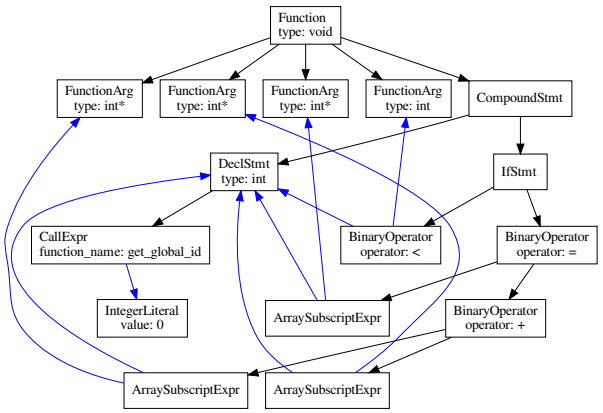

Figure 3. An example kernel as an AST with overlayed dataflow edges (in blue).

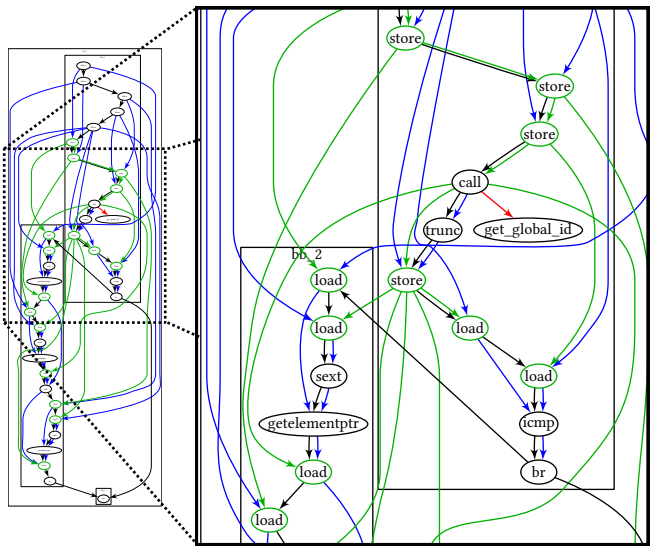

Figure 4. An Example Kernel as a CDFG. The control flow is depicted by black arrows, the data flow by green ones, the red arrow represents an external function call.

sequential nature of strings. As discussed before, for machine learning, this difference can be an advantage or a drawback, e.g. as the ordering might indirectly encode some of the thought processes of the programmer as well. The structure of an AST reflects the grammar of the language, and it would be plausible to think that this exposes the semantics of the code more directly, although it is not obvious that it should. Based on the AST, we define the following representation:

Definition 1 (AST+DF). *A dataflow-enriched AST graph is a labeled digraph, where nodes are labeled as Declarations, Statements, and Types as in the Clang AST. The edges are labeled either as type AST, representing child-of relationships within the AST, and dataflow, representing use use-def relationships for variables.*

To create dataflow-enriched AST graphs, we first extract the raw Clang AST using a tool implemented in the Clang library. As a second step, we reduce the graph by eliminating regular nodes that don't provide additional, structure-critical information. A small graph diameter is generally a desirable property because the information has a limited outreach in the propagation scheme of the graph model introduced in section 4. Specifically, we eliminate nodes of type `DeclRefExpr` and `ImplicitCastExpr` by merging them with their AST-edge successors.

3.3 Control- and Dataflow Graph

Finally, the control and data flow graph (CDFG). This graph organizes the statements in the code not by their grammar, but by the semantics of the possible flow of control in the program. The basic structure of the CDFG is that of a graph. This graph will usually contain cycles, e.g. if the code contains loops. Similar to the AST+DF, we define the following representation:

Definition 2 (CDFG+CALL+MEM). *A LLVM-based control- and dataflow graph enrichted with calls and memory nodes is a labeled digraph, where the nodes are labeled with LLVM IR instructions. The edges of the type control- and dataflow define the basis of the CDFG. The dataflow edges represent operator relationships within the LLVM IR. CALL edges are added based on dependencies to return values of functions. MEM edges represent store-load dependencies to specific memory locations.*

We create the CDFG by using a custom LLVM pass that extracts all LLVM IR instructions, the operands, and the memory accesses. We disable all optimization passes by using optimization level 0 before running our pass.

Figure 4 shows the CDFG representation of the example kernel. Note that the statements in the figure are not in C, but in LLVM IR instead. As can be seen in the figure, this representation includes many new low-level operations, like memory management. It has a different structure, closer to how the calculation is executed than to what is calculated. Thus, this is the representation that is closest to the execution semantics of the program and furthest away from how the programmer expressed a computation. Again, for machine-learning, this poses a trade-off: while exposing the semantics more directly might be beneficial to learn said semantics, it may hinder learning subtleties that are encoded in the human components of a piece of code.

4 Machine Learning Architectures

In order to leverage the compiler-based representations outlined in Section 3, we need deep learning models appropriate to their structure. In this section, we introduce a concrete

architecture to leverage the CDFG and AST representations. As this paper focuses on predictive models, we first explain some core principles for ML-driven predictive models in compiler tasks. After that, we dive into the specifics of the ML architecture required to leverage graph models in the context of predictive models.

4.1 Predictive Models

Predictive models revolve around a simple principle: based on some input data X, predict some output data Y. In the context of compiler optimizations, the input data X is the program code, and perhaps additional information about the task. For example, the task of deciding whether a kernel should be executed on a CPU or a GPU depends on the problem size, which is not part of the kernel's code, and should be considered additionally as input, besides the code.

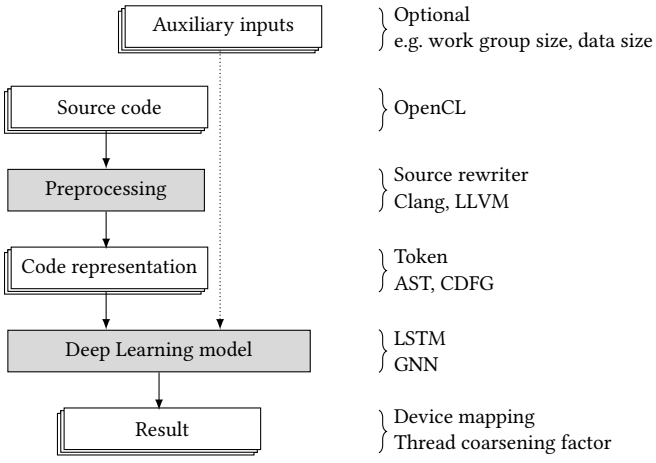

Figure 5. High-level overview of the predictive task.

An end-to-end optimization flow for code-based tasks was proposed in [5], and we base our proposal on it by replacing the representation-specific part with a graph-based deep learning model. Figure 5 depicts the general flow within these predictive models. As can be seen, the input source code is transformed to the appropriate representation for machine learning, using well-established compiler techniques. This includes, for instance, the code normalization discussed in Section 3, the generation of a token string, the syntax analysis producing an (enriched) AST, or a pass creating the CDFG. The output of the flow depends on the concrete predictive task. It could be a simple binary decision like CPU or GPU, or a transformed version of the input program. In this paper we restrict the output to a choice of n possible outcomes, e.g. hardware mappings ($\{\text{CPU}, \text{GPU}\}$) or thread coarsening factors ($\{2, 4, \ldots, 32\}$).

For the deep learning component in sequential-models, as the token-based model used in [5], well-known recurrent neural network architectures like LSTM [8] can be used as representation models. For graph-based models, however, we need to replace the ML architecture used in the flow. In the following, we describe the architecture of a deep learning component of this flow based on graph neural networks. We use this architecture to perform predictive tasks on the AST and CDFG structure.

4.2 Graph Neural Networks

Figure 6 illustrates the deep learning part of our predictive model for graphs with its corresponding components and their relationships. As mentioned above, we base our proposed architecture on the model described in [5] and extend it with the GNN proposed in [13]. Our architecture consists of an embedding layer for creating initial embeddings, a propagation layer for enriching the initial embeddings with structural information, and a prediction layer that aggregates the propagated embeddings and performs a prediction.

The input structure to the model is a labeled graph $G = (V, E)$. The output of the model is an n-sized vector representing a probability distribution, with n being the number of classes. A node embedding vector $h_v \in \mathbb{R}^d$ is assigned to all $v \in V$, where $d \in \mathbb{N}$ is its dimension.

We transform the graph representations of code into the input structure of the model by generating annotated types, considering the graphs of the whole dataset. For the AST-based representation, node types are tuples of (Clang AST node type, property), whereas considered properties are

- the data type for Clang AST nodes of type `Function`, `FunctionArg`, and `DeclStmt`,
- the operator for Clang AST nodes of type `BinaryOperator`,
- and function names for Clang AST nodes of type `CallExpr`.

For the CDFG graph, node types are directly mappable to the LLVM IR node types. Additionally, we consider function names as node types.

Each distinct tuple of the AST graph, or each node type for the CDFG graph respectively, results in a different node type $v_t \in \mathbb{N}$ in the scope of the graph model. For each edge, we also add an edge with the reverse direction, in order to improve the propagation capabilities of the model.

Initial Embedding Layer Nodes from the input graph are represented as one-hot encoded vectors, i.e. vectors $(e_j)_i = \delta_{i,j}$, where $\delta_{i,j}$ is the Kronecker Delta which is 1 iff $i = j$ and 0 otherwise. Since the number of possible node types tends to be very large in practice, we introduce this embedding layer to reduce the dimension of these one-hot encoded node vectors, to the smaller size $d \in \mathbb{N}$ described above. The node embedding vectors h_v are computed by applying a learnable function $f_{\text{init}}(v_t)$ to the node annotation vector v_t. The learnable function is implemented as a Multi-Layer Perceptron (MLP) neural network.

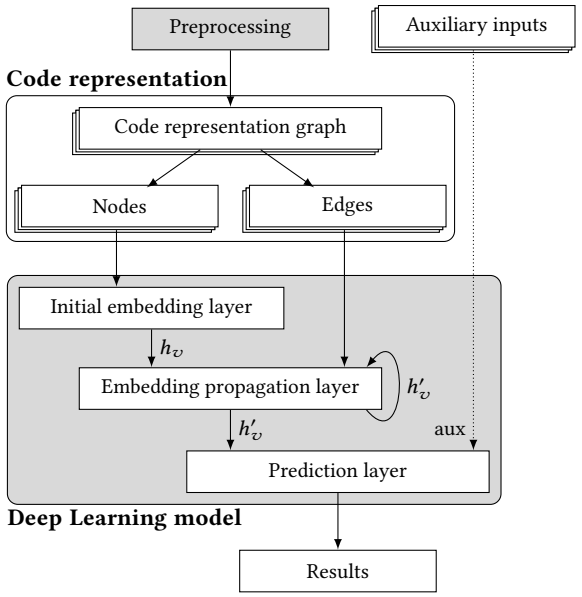

Figure 6. An overview of the predictive model architecture based on graph neural networks.

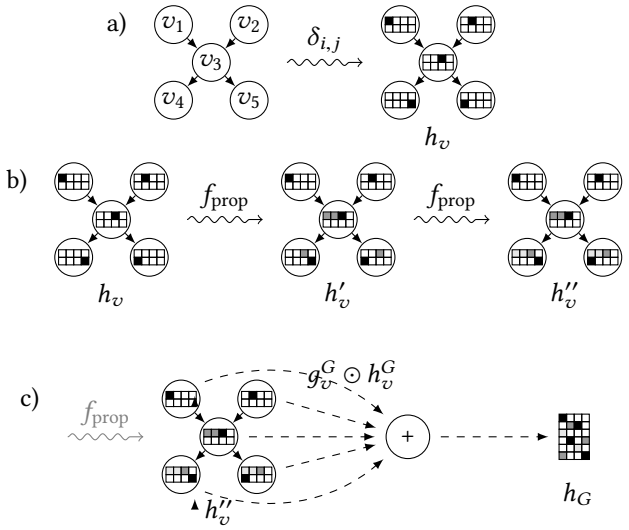

Figure 7. An illustration of a) encoding the initial node embeddings, b) the embedding propagation scheme with 2 iterations, and c) the aggregation to a graph embedding.

$$h_v = f_{\text{init}}(v_t).$$

Embedding Propagation Layer With the initial encodings based only on the node type, the encoding does not contain any structural information of the graph, i.e. as expressed by the edges. By applying an iterative information propagation scheme, the initial node embedding vectors produced by the embedding layer are enriched with this structural graph information. For a fixed number of iteration steps T, nodes propagate their embeddings to their directly adjacent neighbors and eventually result in embeddings that contain information about a T-sized neighborhood. This is illustrated in Figure 7, where the colored cells represent the information on the nodes encoded in the embedding vectors.

In each iteration, messages are formed for each node and each edge by using a learnable function $f_{\text{msg}}(h_v, e_t)$ with the embedding information h_v and the edge type information e_t. It consists of learnable parameters A and b for each edge type e_t. When all messages of an iteration have been passed to their target nodes, they are aggregated per target node and new intermediate node embedding vectors h'_v are formed by applying a learnable function $f_{\text{prop}}(a_v, h_v)$. This function is implemented as a Gated Recurrent Unit (GRU), a variant of a recurrent neural network with similar performance to an LSTM, but with less learnable parameters [4]. After T iterations of the scheme, the node embedding vectors reach a final state. As depicted in the figure, they are then aggregated to a single graph embedding h_G, which happens in the prediction layer.

$$f_{\text{msg}}(h_v, e_t) = A_{e_t} \cdot h_v + b_{e_t},$$
$$a_v = \sum_{u:(u,v)\in E} f_{\text{msg}}(h_v, e_t), \qquad h'_v = f_{\text{prop}}(a_v, h_v)\ \forall v \in V.$$

Prediction Layer This layer maps the final node embedding vectors to a probability distribution. To achieve this, a fixed-size graph embedding vector is computed by aggregating the final node embedding vectors and then mapping the graph embedding vector to the output.

The aggregation is implemented by summing all final node embedding vectors, after mapping them to a higher size with the learnable functions $f_m(h_v)$ and $g_m(h_v)$. An attention mechanism, implemented by the element-wise product $f_m(h_v) \odot g_m(h_v)$ of $f_m(h_v)$ and $g_m(h_v)$, decides the relevance of individual nodes in the current task. The functions are implemented as MLPs with hyperbolic tangent and sigmoid activation. The sigmoid activation function is a natural fit for the attention mechanism because it outputs a range of $(0, 1)$ that is multiplied element-wise with the output vector of $f_m(h_v)$. The hyperbolic tangent function on the other hand, normalizes the values to a range of $(-1, 1)$ within the model, yielding better performance.

A final learnable function $f_{out}(h_G)$ computes the output based on the graph embedding vector h_G that is optionally concatenated with the auxiliary inputs *aux*. It is implemented as a MLP with a softmax activation function.

$$h_v^G = f_m(h_v), \qquad g_v^G = g_m(h_v),$$
$$h_G = \sum_{v\in V} g_v^G \odot h_v^G, \qquad \text{out} = f_{\text{out}}(h_G, \text{aux}).$$

5 Evaluation

In this section, we present an evaluation of our methods using the different representations of code.[2] We do this by using two different complex tasks on OpenCL kernels. These tasks require an understanding of the full kernel, how different parts of it interact and the subtle interplay of them with the memory hierarchy and SIMD capabilities of the different target platforms. As such, the tasks serve as a good first evaluation of the representations, as we expect only a representation capable of capturing these intricacies to perform well on them.

For both tasks, we reproduce[3] the results of [5] and [3] and integrate our method into the experimental setup. In this setup, the dataset is split into k parts which are used to make different training and testing sets in a k-fold cross-validation scheme. In this scheme, the model is iteratively trained anew using $k-1$ parts of the dataset as training data and is subsequently tested on the one part that was left out while training. This is repeated for all $k-1, 1$ partitions.

In these experiments, we observe a variance in the results for all models, as the initialization of their weights is random and the model converges to different minima. Therefore, we repeat the execution of each experiment 10 times and report aggregated results.

5.1 CPU/GPU Mapping

The problem of CPU/GPU mapping considers OpenCL kernels and has the goal of classifying them as CPU or GPU, depending on where they can be expected to run faster. This problem has been studied extensively, and several approaches have been proposed to solve it. For example, Grewe et al. [17] propose a heuristic based on a decision tree to solve this. Cummins et al. [5], on the other hand, used a machine learning approach based on an LSTM and a token representation to improve upon this. More recently, Ben-Nun et al. [3] used a model based on finding good embeddings for LLVM instructions to improve this problem further. We compare our compiler-graph-based models to these approaches.

For evaluating the models, we want to asses their generalization performance. In the tasks we use the accuracy as a metric, which is the ratio of the correctly predicted device mappings over all predicted device mappings. As an additional metric for the performance benefit that the methods bring we use the speedup. A correctly predicted mapping of a sample yields a faster execution, resulting in a speedup over a static mapping. In a static mapping, a single platform (CPU/GPU) is selected and all kernels are mapped to this platform. To select, the platform which is fastest in most of the training samples for all test benchmarks is chosen.

[2]Artifact available at https://github.com/tud-ccc/learning-compiler-graphs

[3]In fact, the results of [5] we show are better than the ones originally published. We could reproduce this behavior in the published artifact of this original work as well, obtaining better results there too when loading the published models.

Additionally, we compare the models in terms of the number of trainable parameters, which describe their sizes, as well as their training and inference time. Inference time is particularly important for compiler-related tasks, as it translates to a faster compilation time for an end-user.

Experimental Setup The dataset consists of the seven benchmark suites AMD SDK, NPB, NVIDIA SDK, Parboil, Polybench, Rodinia, and SHOC along with the execution times for both CPU and GPU on two different heterogeneous systems, one with an AMD Tahi 7970 GPU and one with an NVIDIA GTX 970 GPU.

The construction of the code representation graphs for the whole dataset leads to 92 node types for the GNN model using AST+DF (here, GNN-AST) and 140 node types for the GNN model using CDFG+CALL+MEM (here, GNN-CDFG). To compute the initial node embeddings, we map the one-hot encoded vectors with an MLP f_{init} consisting of 2 hidden layers of size 64 to vectors of size 32. As propagation size T, we chose 4 iterations, which yields embeddings that include a 4-neighborhood of the individual nodes. The graph embedding vector h_G of size 64 is created by aggregating the node embeddings using the two MLPs f_m and g_m, which are dimensioned with 2 hidden layers of size 64. The graph embedding is mapped with an MLP with two hidden layers of size 32 to a dimension of 32, then serves as input to the prediction model used in [5] and [3], which is a MLP with 1 hidden layer of size 32.

We compare both of our methods to the state of the art methods DeepTune [5] and inst2vec [3]. We also compare it to the decision-tree-based model of Grewe et al [17]. Additionally, we compare it to the static mapping model, and to a model choosing CPU or GPU at random.

Table 1 gives a comparison of the network sizes, in terms of the number of trainable parameters, as well as the training and inference times. We can see that Grewe et al's model is considerably faster in training and inference and has the least amount of trainable parameters. The deep learning models are larger in size by several orders of magnitude. It is important to note that the AST-based model is an order of magnitude faster in inference than the other deep-learning based models.

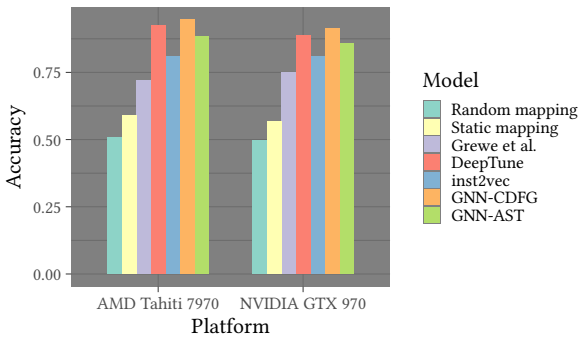

Figure 8. Accuracy results compared to state-of-the-art.

Table 1. Model sizes for the device mapping task

Model	Code Representation	Model architecture	No. of learnable parameters	Training time	Inference time
Grewe et al. [17]	Manual features	Decision tree	14	1.02×10^{-3} s	8.46×10^{-5} s
DeepTune [5]	C tokens	Recurrent neural network	76908	6.72×10^{2} s	9.02×10^{-1} s
inst2vec [3]	LLVM-IR tokens	Recurrent neural network	649372	1.08×10^{3} s	1.34 s
GNN-AST	AST+DF	Graph neural network	139846	8.40×10^{2} s	9.95×10^{-2} s
GNN-CDFG	CDFG+CALL+MEM	Graph neural network	89798	1.59×10^{3} s	9.8×10^{-1} s

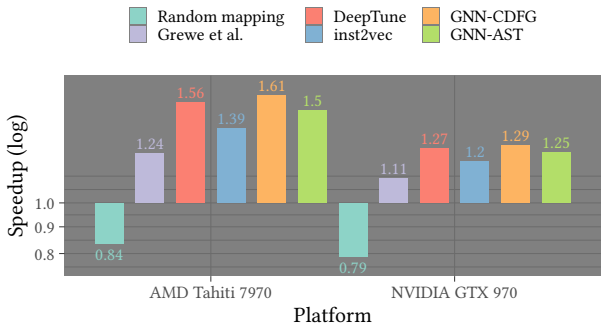

Figure 9. Speedups (geometric mean) compared to state-of-the-art.

Figure 8 shows the overall accuracy results on the different benchmarks. We see how our graph-based models perform better than `inst2vec`, albeit modestly. While `DeepTune` produces a slightly better accuracy than `GNN-AST`, `GNN-CDFG` yields the highest accuracy overall. Similarly, Figure 9 shows the speedups (geometric mean over all executions) [4]. We see how the trends of the speedup results are comparable to those seen from the accuracy results, which is to be expected as the two metrics are linked by their definitions.

While useful for comparison with the state-of-the-art, the setup used by [3, 5] trains the model with kernels from the same benchmarks that it then uses to evaluate the heuristic. Having 7 different benchmark suites, we believe that using these as groups in a k-groups-split methodology yields more insight about the generalization capabilities of the models. This way, they are tested on kernels from a benchmark suite they have not been trained on. To this end we split the dataset into 7 parts, each of the 7 parts being the different benchmarks suites, instead of 10 randomly-chosen parts out of the set of all kernels of the benchmark suites. This way, the model is tested on data that is fundamentally different than the benchmarks its was trained on.

Figure 10 shows the results of the experiment with this alternative setup. Since we split the train and test sets by benchmarks, we can see how the models fare on every benchmark after being trained on the other six. It is notable how the different methods can have vastly different results on the different benchmarks. The final entries for each platform show an aggregated result over all benchmarks (artithmetic mean).

We can see that GNN-AST is not only the one with the best overall results, but also the most consistent ones. It had an overall accuracy of 60.6%, which is over 12 percentage points better than that of DeepTune at 47.9% and even better than the 44.9% overall accuracy obtained by inst2vec. In this case, GNN-CDFG had a similar performance, beating DeepTune by less than a percentage point in accuracy at 48.5%. In fact, we see how when tested on different benchmark suites than they were trained on, all state-of-the-art methods we compared to here perform worse than a coin-toss (50.9% accuracy). This indicates that the models probably do not learn the relationship between the code's semantics and the optimal compute device in a way that is generalizable when the code becomes different enough.

5.2 Thread Coarsening

In parallel architectures, faster executions can be achieved by merging multiple parallel threads in certain situations. The thread coarsening factor is a parameter that controls this behavior in OpenCL. Various predictive models have been proposed to solve this task, such as that by Magni et al [15], which uses an MLP neural network based on static code features, such as the quantities of certain instructions on the LLVM IR. More recently, Cummins et al. [5] and Ben-Nun et al. [3] proposed to use LSTM-based deep learning models based on a source- and LLVM IR-level sequence of tokens. The dataset for this problem consists of 17 selected kernels from the AMD SDK, NVIDIA SDK, and Parboil benchmark suites. As output, the flow decides among 6 classes for every kernel, corresponding to coarsening factors of 1, 2, 4, 8, 16, 32. We reproduced these models and the corresponding results and used these as baseline to compare to our methods.

For this dataset we get a total of 46 distinct node types for GNN-AST and 54 node types for GNN-CDFG. In this task, we keep the dimensions of the model at a minimum, as the amount of training data is quite slim. We apply one-hot encoding to the nodes represented as node types and map the resulting vectors with the MLP f_{init} to a size of 4. This MLP only contains the input and output layers and no hidden layers. After 4 propagation time steps T, we aggregate the

[4] Our reported values differ from the original publications of the baseline methods, because we aggregate using the geometric mean instead of the arithmetic mean.

Table 2. Model sizes for the thread coarsening task.

Model	Code Representation	Model architecture	No. of learnable parameters	Training time	Inference time
Magni et al. [15]	Manual features	MLP		8.63 s	2.62×10^{-4} s
DeepTune [5]	C tokens	Recurrent neural network	76838	8.66×10^{-1} s	5.59×10^{-1} s
inst2vec [3]	LLVM-IR tokens	Recurrent neural network	649030	1.92×10^{1} s	2.08×10^{-1} s
GNN-AST	AST+DF	Graph neural network	914	4.83 s	1.21×10^{-3} s
GNN-CDFG	CDFG+CALL+MEM	Graph neural network	1024	5.21 s	1.32×10^{-3} s

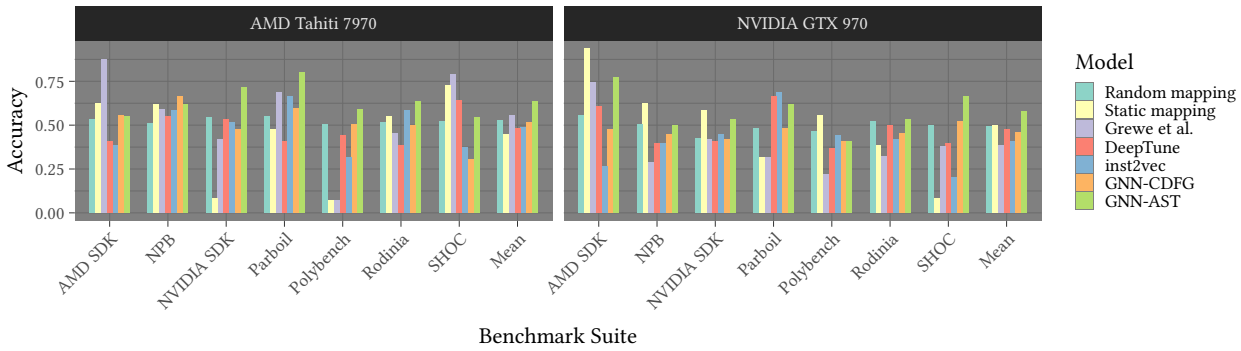

Figure 10. Accuracy of device mapping in grouped setting.

node embedding vectors to a graph embedding vector h_G of size 8 by using the two MLPs f_m and f_g with no hidden layers. We use the prediction model used in [5] and [3], which consists of one hidden layer, but with a reduced dimension of 4 instead of 32. All baseline models we trained in the reported configurations.

Again, we compare the model sizes and training and inference times of the different models. The results can be seen in Table 2. And again here we see how inference in GNN-based models is orders of magnitude faster than the LSTM-based sequential counterparts.

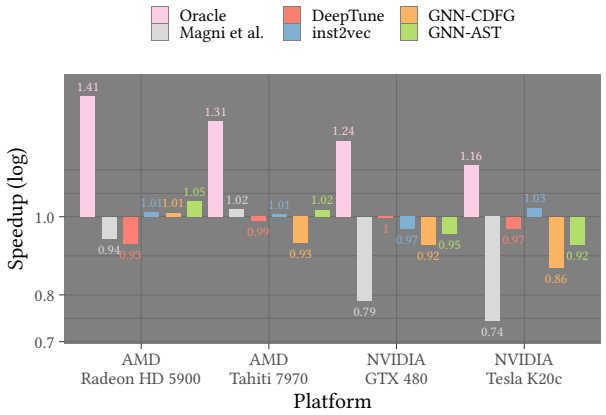

Figure 11. Speedup results in thread coarsening task.

Figure 11 shows the results of the thread coarsening experiment. In the figure we add an "Oracle" for reference, which depicts the best possible speedups. As we can see in the figure, the sequence-based models perform better than both graph-based models in three of the platforms. It is notable, however, that in most cases the predicted thread-coarsening factors yield an overall slowdown. Overall, DeepTune yields an overall speedup (geometric mean) of around 0.97 across all platforms, GNN-AST is slightly better with 0.98 and GNN-CDFG is slightly worse with 0.93. The Magni et al. model yields an overall speedup of 0.87. The best results in this task are achieved by inst2vec, which has an overall speedup of 1.00, i.e. just as good as doing nothing. In general, all deep learning methods fare comparably bad at this task. This might be explained in part by the modest possible maximal speedups, which cap at 1.28 overall. However, it is perhaps also an indication that current deep learning models of code are not yet up to this task.

5.3 Comparison of Graph-Based Models

The graph-based representations we have used for deep learning in this paper feature several enhancements, like the dataflow edges enriching the AST. Similarly, for the CDFG we included edges connecting the corresponding instructions for function calls and the data in load and store instructions. In this section, we want to use the three experimental setups described above to compare how our model fares with and without these enhancements. For this, we used two versions of our AST-based representation, with (AST+DF) and without the dataflow edges (AST). Similarly, for the CDFG representation, we considered four different versions: just control flow edges (CFG), control and dataflow edges (CDFG), CDFG+CALL with call edges, and CDFG+CALL+MEM.

We trained the model using similar configurations as in the previous experiments. Figure 12 shows the results with the

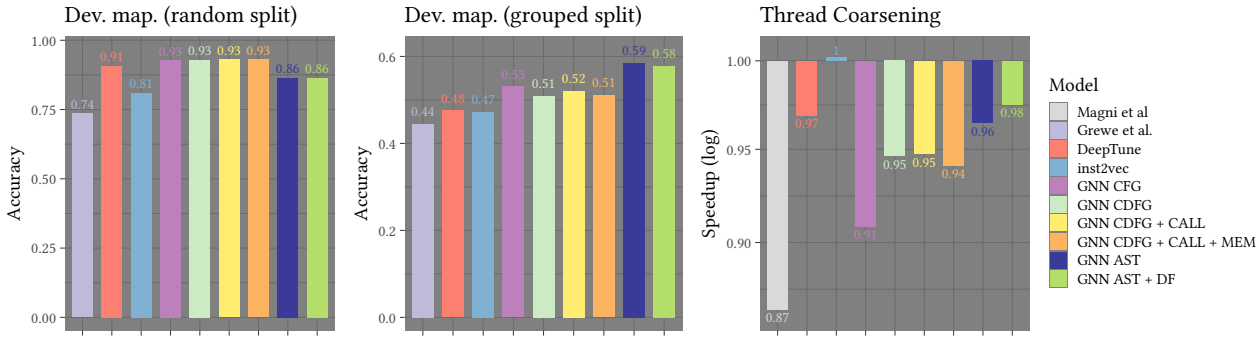

Figure 12. Comparison of graph-based models with different enhancements.

different versions of our graph-based representations. In the random setup for device mapping, all variants of the CDFG-based representation fared similarly well, yielding the best results overall. In the grouped setup, we see how the AST-based representation yields better results. All experiments, the grouped split mapping especially, show that the addition of the dataflow edges to the AST was fruitful. Finally, as discussed above, the results on the thread coarsening task are pretty modest across all deep learning models.

From these experiments, we cannot conclude that there is a single best compiler-based representation of code for deep learning models. In the random split of the mapping task, where the training data resembles more the test data, the models closer to the execution semantics (CDFG) had the best performance. On the other hand, with the grouped split, where a higher degree of generalization is required, the AST with its more abstract semantics, closer to the programmer, performed better. For the thread coarsening task, which has very little data points available, the non-end-to-end approach of inst2vec fared best, albeit also not particularly well, yielding no speedup overall.

6 Conclusion

In this paper, we have compared code representations for deep learning based on the representations used in the compiler: sequences of tokens, abstract syntax trees, and control and data flow graphs. We have shown different graph-based methods to be better suited for a complex classification task, identifying the optimal CPU/GPU mapping for OpenCL kernels. In particular, we outperformed state-of-the-art approaches using sequential models based on token sequences. We also showed more generalization capabilities by splitting training and test data into sets coming from disjoint benchmark suites instead of random subsets of all kernels. In this alternative setup, our graph-based model GNN+AST performed significantly better than their sequential counterparts, which failed to beat a coin-toss.

From our experiments, we cannot conclude that there is a single best compiler-based representation of code for deep learning models. In the random split of the mapping task, where the training data resembles more the test data, the models closer to the execution semantics (CDFG) had the best performance. On the other hand, with the grouped split, where a higher degree of generalization is required, the AST with its more abstract semantics, closer to the programmer, performed better. For the thread coarsening task, which has very little data points available, the non-end-to-end approach of inst2vec fared best, albeit also not particularly well, yielding no speedup overall.

The CPU/GPU classification task is a complex task where the interaction between different parts of the code and the target architecture all play a role. Thus, this task serves as a good first challenge for compiler analysis methods. However, the related predictive task of finding optimal thread coarsening factors for OpenCL kernels proved to be too challenging for both sequential and graph-based models. The results of this task indicate that while useful, deep learning methods still struggle on complex compiler-related tasks. Note that the dataset for this task is small, which poses an additional challenge to deep learning models.

We believe our results encourage further investigation of compiler-based representations of code. Programming languages and machine semantics being strict and structured as they are should also be an advantage in terms of learning, and we can take advantage of decades of research into compiler methods to improve machine learning methods in this domain. Furthermore, OpenCl kernels are comparatively small. In future work, we plan to investigate these methods on larger code fragments, where we believe the graph structures should be even better suited to track long-range dependencies between parts of the code.

Acknowledgments

We thank Chris Cummins and Hugh Leather for making the baseline methods and datasets available, as well as for valuable feedback on the approach. We also thank Christian Menard for an initial version of the LLVM pass. Further, we thank the Center for Information Services and HPC (ZIH) at TU Dresden for providing computation resources. This work was supported in part by the German Research Council (DFG) through the TraceSymm project CA 1602/4-1 and the Studienstiftung des deutschen Volkes.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[3] 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*, pages 3585–3597, 2018.

[4] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[5] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2017)*, September 2017.

[6] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[7] A. Goens, A. Brauckmann, S. Ertel, C. Cummins, H. Leather, and J. Castrillon. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, MAPL 2019, pages 38–46, New York, NY, USA, June 2019. ACM.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] D. D. Johnson. Learning graphical state transitions. In *ICLR*, 2017.

[10] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[11] H. Leather, E. Bonilla, and M. O'boyle. Automatic feature generation for machine learning–based optimising compilation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):14, 2014.

[12] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

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

[14] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[15] 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*, pages 455–466. ACM, 2014.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[17] M. F. O'Boyle, Z. Wang, and D. Grewe. 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)*, pages 1–10. IEEE Computer Society, 2013.

[18] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[19] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[20] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In A. Mycroft and A. Zeller, editors, *Compiler Construction*, pages 185–201, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[21] Z. Wang and M. OâĂŹBoyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[22] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.