ComPy-Learn: A toolbox for exploring machine learning representations for compilers

Alexander Brauckmann, Andrés Goens and Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden, Germany
{first.last}@tu-dresden.de

***Abstract*—Deep Learning methods have not only shown to improve software performance in compiler heuristics, but also e.g. to improve security in vulnerability prediction or to boost developer productivity in software engineering tools. A key to the success of such methods across these use cases is the expressiveness of the representation used to abstract from the program code. Recent work has shown that different such representations have unique advantages in terms of performance. However, determining the best-performing one for a given task is often not obvious and requires empirical evaluation. Therefore, we present ComPy-Learn, a toolbox for conveniently defining, extracting, and exploring representations of program code. With syntax-level language information from the Clang compiler frontend and low-level information from the LLVM compiler backend, the tool supports the construction of linear and graph representations and enables an efficient search for the best-performing representation and model for tasks on program code.**

***Index Terms*—Compilers, Clang, LLVM, Machine Learning, Code Representations**

I. Introduction

In the last decades, improvements in processing power have enabled a revolution in machine learning (ML). ML methods have had an impact in most disciplines of engineering, and compilers are not the exception [1]. While perhaps not yet achieving the same level of success as in image recognition, ML methods have been used in multiple ways in the context of compilers and programming languages [2].

While earlier methods rely on manually-defined features, in many domains they have been outperformed by deep learning methods [3]. This holds also true for compiler tasks [4]. Independent of the goal, any deep learning method that operates on source code has an underlying concrete representation of the code. The taxonomy presented in [2] distinguishes between the representation itself, and the ML model used for the representation. For example, one of the most common representations are characters or tokens, which can be modeled in different ways, e.g. with simple n-grams, or deep neural network models like Long Short Term Memory (LSTM). However, other representations can be advantageous to expose different structures in the code, like those based on the language syntax or on control- and dataflow graphs.

The representation chosen and the corresponding model play an important role in the effectiveness of ML methods in compilers. They determine what information is available to the algorithm, as well as how accessible it is, based on the structure of the representation [5]. However, in order to use

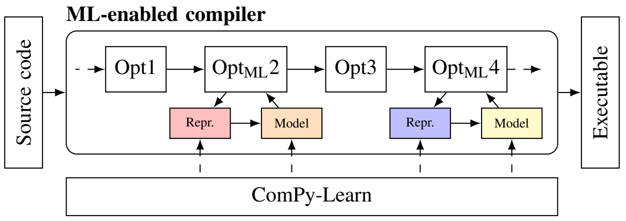

Fig. 1. ComPy-Learn in the context of a ML-enabled compiler.

these ML methods in practice in a compiler, we need to embed them into one. Figure 1 shows our vision of such an ML-enabled compiler. Multiple different optimizations will benefit from different models. We first need to understand which representations and models are best-suited to the individual optimizations in these compilers. For a single optimization already, this is a complex and time-consuming endeavor, since modifying the representation and the model requires a significant rewrite of the codebase of a learning compiler. In practice, integrating multiple optimizations leveraging different representations would be prohibitively costly to engineer without a framework to explore and manage the different representations and models.

In this paper we present a toolbox to effectively explore multiple representations and models for code in ML-enabled compilers. Our toolbox, ComPy-Learn[1], fills an important gap needed to research and design efficient ML-enabled compilers. Its modular design clearly separates the learning task, the representation and the model (cf. Section II). This allows designers to seamlessly explore multiple models and representations to find the best-suited ones for the task at hand by using a simple programming model (cf. Section III). We evaluate ComPy-Learn by reenacting the exploration carried out by independent groups over the past years for ML-enabled OpenCL kernel mapping (cf. Section IV).

II. Architecture and Design

The high-level flows of ComPy-Learn, its components, and the ML Pipeline are shown in Figure 2: Given a set of (Source Code, Property) pairs, commonly referred to as the dataset, ComPy-Learn learns a mapping function $f(\text{Source Code}) \longrightarrow \text{Property}$ with the objective to best fit the training

[1]Tool available at https://github.com/tud-ccc/compy-learn

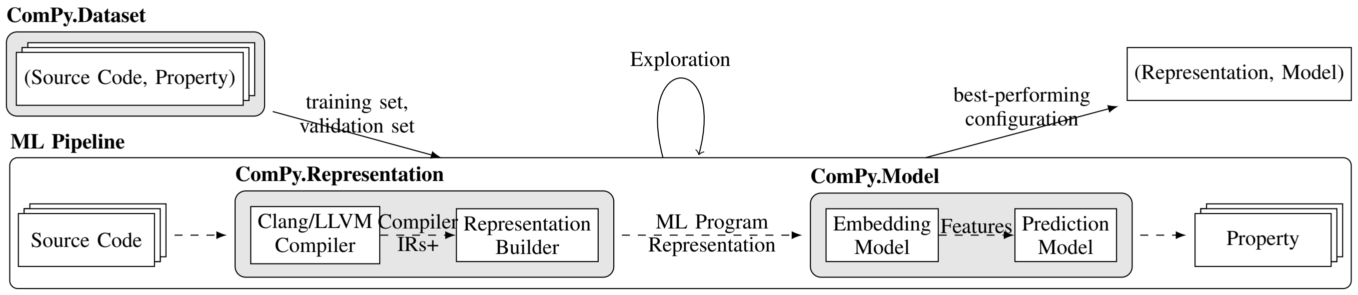

Fig. 2. High-level flow of ComPy-Learn and ML pipeline. ComPy-Learn flow in solid, ML Pipeline flow in dashed arrows.

set. Multiple such functions are *explored* by evaluating on different combinations of representations and models in the *ML Pipeline*. Finally, ComPy-Learn results in a best-performing configuration of *Representation* and *Model*. In the ML Pipeline flow, the *ComPy.Representation* components first transform the raw *Source Code* into a *ML Program Representation*, which is input to components of *ComPy.Model* that predict the properties.

A. Program representation

The program representation component consists of Clang/LLVM-Compiler tools and a representation builder that constructs the ML program representations:

The *Clang/LLVM compiler* tools are used to capture compiler-internal intermediate representations (IRs) of a given program and construct a data structure holding this information. The IRs supported at the time of writing are based on the Clang Abstract Syntax Tree (AST) and the LLVM IR in sequential and graph-based variants. The IRs are enriched with more semantics from compiler analyses and therefore called IRs+. The sequential IRs+ are identifier-normalized, similar to [4], [6], which results in identifiers of functions, constants, and variables being named in a fixed, predefined scheme. In addition to the raw token information of [4], ComPy-Learn's sequential IRs+ contains token kind information, allowing to construct custom ML program representations. The graph-based variants are enriched with additional information from compiler-internal analyses, allowing to construct representations with e.g. memory access, control- and dataflow information, such as the ones presented in [5], [7].

The *representation builder* component is used to transform the IR+ to ML program representations. Within the program representation builder, multiple definitions of representations exist and can be exchanged in this pipeline, allowing for reusability.

This two-component design is motivated by the observation that multiple ML program representations use common properties of the compiler IRs. This approach also allows for extensibility because new code representation structures can be built conveniently by traversing the IR+ objects without modifying the implementation-heavy Clang- and LLVM tools that are used to extract information. A detailed example of this is shown in Section IV.

B. ML model

To make predictions on a code sample, after encoding it in an ML program representation, an ML model is optimized to fit the training set. This model consists of two parts, a learnable embedding model that extracts features, and a prediction model that outputs a property from the extracted features.

Different *embedding models* are supported, depending on the structure of the ML program representation. Recurrent Neural Network (RNN) models, on the one hand, learn features of linear nature by iteratively passing a hidden state, as well as token information along a sequence [8]. Various variants of this model class are supported as off-the-shelf models in the underlying Deep Learning frameworks TensorFlow and PyTorch, and can be configured by the user. Graph Neural Networks (GNN) models, on the other hand, learn features of graph structures by iteratively passing hidden states along a graph's edges [9]. ComPy-Learn integrates with the popular GNN implementation frameworks [10] and [11]. Part of these frameworks is e.g. the popular Gated Graph Neural Network (GGNN) architecture that has proven to be well-suited for learning tasks on source code [5], [7], [12].

The *prediction model* can be configured to perform classifications or regressions, supporting a variety of predictive graph- and sequence-level classification tasks. Node- and token-level prediction tasks are not supported currently, but can be easily added if a use case requires so.

C. Dataset

Not being part of the ML pipeline per se, the *ComPy.Dataset* component provides tools to define a dataset, consisting of code samples and prediction target properties.

III. Programming Model

We will demonstrate relevant parts of ComPy-Learn's API on the highest level. The learning pipeline, as shown in Listing 1, consists of the stages of defining the representation and model combinations to be explored. Then, for each combination it builds the representation, trains the model, and finally, evaluates the performance on the test set.

In lines 1-4 we define various combinations of representations and models. Each of the combinations consists of a triple of subclasses of `RepresentationBuilder`, `Visitor`, and `Model`. The builder is responsible of constructing the IR+ on which the Visitor extracts linear or graph structures.

```
1  combinations = [
2      (ASTGraphBuilder, ASTDataVisitor, GnnModel),
3      (LLVMGraphBuilder, LLVMCDFGVisitor, GnnModel),
4      (SyntaxSeqBuilder, SyntaxSeqVisitor, LSTMModel)]
5
6  summaries = []
7  for builder, visitor, model in combinations:
8    clang_driver = ClangDriver(language=C,
9                               optimization_level=O3)
10   samples = DevMapDataset.preprocess(
11       builder(clang_driver),
12       visitor)
13
14   summary = model.eval(samples[train_idx],
15                        samples[test_idx])
16   summaries.append(summary)
17
18 report(summaries)
```

Listing 1: Example of an exploration.

In lines 6-12 we build each of the representations defined in the combination triples. For this, the builder relies on a `ClangDriver` object that represents the functionality of the Clang compiler frontend. This object is configurable with e.g. the programming language, optimization level, and include paths. A subclass of `Dataset` that implements a task-specific dataset then accepts a builder and a visitor. The builder is constructed with the just instantiated `ClangDriver`.

In lines 14-16, the model is trained on a training set and continuously evaluated on a test set. The resulting summaries contain the accuracies of the training set and test set for each epoch for each combination, which enables further evaluation. Finally, a function `report` prints the performance of the configurations.

The framework's API offers the user to customize the pipeline by allowing to define custom combination triples. User-implemented objects conforming to the `RepresentationBuilder`, `Visitor` and `Model` interfaces can be passed to the pipeline.

IV. Evaluation

To show ComPy-Lean's capabilities to construct ML-program representations, we will re-implement several representations from literature. As a compiler optimization for demonstration, we use the binary classification task of device mapping, initially presented in [13]. The optimization consists of deciding where to best place an OpenCL kernel on a system with two alternative devices. A correct mapping leads to a faster execution time compared to a wrong mapping. In ComPy-Learn's terminology, the property to predict is {CPU, GPU}.

Figure 3 shows visualizations of several of the representations on the example of the recursive factorial function shown in Figure 3a). The representations in Figures 3d) - 3f) have been created using the visitor shown in Listing 2 through traversal of the LLVM IR+. The code regions that account for different edges in the graph representations are highlighted in the corresponding colors. Various existing representations from bleeding-edge published work, such as [5], [7] and beyond can be easily constructed from this common data structure (IR+), as demonstrated in Figure 3. Further, we use the pipeline of ComPy-Learn to explore various combinations of program representations and ML models on the described device mapping optimization. Note that this evaluation shows how we can construct and use multiple representations; it is not meant as a conclusive comparison of their accuracies. In particular, we evaluate on a single dataset split instead of k-fold cross-validation as in the original works and report the results of only one experiment execution.

TABLE I
Evaluation results of different representations and models.

Ref	ML program representation				Model	Acc
	IR+	***Structure***	***Nodes***	***Edges***		
[4]	AST	Seq	C tokens	next	LSTM	0.79
	AST	Graph	AST stmts	ast	GGNN	0.79
[5]	AST	Graph	AST stmts	ast, use-def	GGNN	0.79
	AST	Graph	AST stmts	ast, use-def, cfg	GGNN	0.79
[6]	LLVM	Seq	LLVM tokens	next	LSTM	0.76
	LLVM	Graph	LLVM IR	control, use-def	GGNN	0.75
[5]	LLVM	Graph	LLVM IR	control, use-def, call	GGNN	0.75
	LLVM	Graph	LLVM IR, Basic Blocks	control, use-def, call, basic block	GGNN	0.81
[7]	LLVM	Graph	LLVM IR, variables, constants	control, use-def, call	GGNN	0.70

For the evaluation, we use the dataset of [4], which consists of 680 OpenCL kernels with their best device mappings as prediction properties in the AMD variant. We split it into a training and validation set. The training set, consisting of 90% of the kernels, is used to train the combinations of representations and models for 1000 epochs each, using the Adam optimization algorithm [14] with a learning rate of 0.001 and batches of 64 samples. We measure the accuracies of the training and validation set after each epoch and report the validation accuracy of the epoch that resulted in the best training accuracy. The results in Table I show that a LLVM graph representation, augmented with Basic Block information yields the highest accuracy in this particular setting. Finding this representation was possible because of the convenient design and exploration capabilities of ComPy-Learn. However, [5] has shown that AST-based graph representations and LLVM IR-based sequence representations are best suitable for other datasets and tasks, making an exploration indispensable.

V. Conclusion

In this paper we presented ComPy-Learn, a toolbox for designing ML-enabled compiler optimizations. Using ComPy-Learn, we were easily able to explore multiple different combinations of representations and models, thereby reproducing recently proposed representations and even beyond. Because of its user-friendliness, modularity and extensibility, we believe it to be an ideal starting point for tasks in this domain.

For future work, we hope that its open-source nature and modular design permit to research compiler optimizations, code representations, and models more easily. For our own future work, we aim to extend ComPy-Learn with further code representations, components for explaining features, and new techniques for model optimization. Further code representations of existing work would allow for a more comprehensive exploration. To investigate the learned features, we plan to

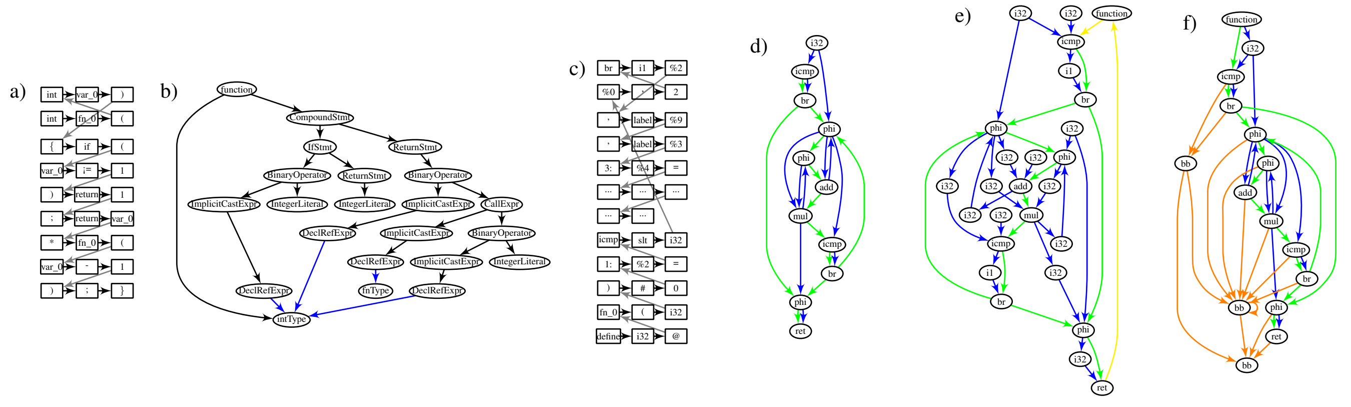

Fig. 3. Example program in a selection of the supported representations: a) Syntax token sequence, b) AST-graph, augmented with use-def edges, c) LLVM IR token sequence, d) LLVM IR Control- and Dataflow graph (CDFG), e) LLVM IR CDFG, augmented with call edges, variable and constant nodes. f) LLVM IR CDFG, augmented with basic block nodes and their belongs-to and control-flow edges. AST edges in black, control-flow edges in green, data edges in blue, call edges in yellow, basic-block edges in orange.

integrate explainability methods into the framework, enabling insights about decision criteria and the discovery of patterns.

Acknowledgments

This work was funded in part by the German Federal Ministry of Education and Research (BMBF) within the project ScaDS AI (BMBF 01IS18026B), the German Research Council (DFG) through the TraceSymm project CA 1602/4-1 and the Studienstiftung des deutschen Volkes. We thank the Center for Information Services and HPC (ZIH) at TU Dresden for providing computation resources.

References

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

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

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-end deep learning of optimization heuristics," in *PACT*, ACM, 2017.

[5] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, pp. 201–211, 2020.

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

[7] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "Programl: Graph-based deep learning for program optimization and analysis," *arXiv preprint arXiv:2003.10536*, 2020.

[8] W. De Mulder, S. Bethard, and M.-F. Moens, "A survey on the application of recurrent neural networks to statistical language modeling," *Computer Speech & Language*, vol. 30, no. 1, pp. 61–98, 2015.

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

[10] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[11] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

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

```
class CDFGCallConstantBBVisitor(Visitor):
  def __init__(self):
    Visitor.__init__(self)
    self.G = nx.MultiDiGraph()

  def visit(self, v):
    if isinstance(v, llvm.graph.Function):
      self.functions[v.name] = v
      self.G.add_node(v, attr=('function'))
      self.G.add_edge(v, v.entryInstr,
                      attr=('call'))
      for arg in v.args:
        self.G.add_node(arg, attr=(arg.type))

    if isinstance(v, llvm.graph.BasicBlock):
      self.G.add_node(v, attr=('bb'))
      for instr in v.instructions:
        self.G.add_edge(instr, v, attr=('bb'))
      for succ in v.successors:
        self.G.add_edge(v, succ, attr=('bb'))

      instr_prev = v.instructions[0]
      for instr in v.instructions[1:]:
        self.G.add_edge(instr_prev, instr,
                        attr=('cfg'))
        instr_prev = instr

      for succ in v.successors:
        self.G.add_edge(v.instructions[-1],
                        succ.instructions[0],
                        attr=('cfg'))

    if isinstance(v, llvm.graph.Instruction):
      self.G.add_node(v, attr=(v.opcode))

      if v.opcode == 'ret':
        self.G.add_edge(v, v.function,
                        attr=('call'))
      if v.opcode == 'call':
        called = self.functions[v.callTarget]
        self.G.add_edge(v, called.entryInstr,
                        attr=('call'))
        for exit in called.exitInstr:
          self.G.add_edge(exit, v, attr=('call'))

      for op in v.operands:
        if isinstance(op, llvm.graph.Arg) or
            isinstance(op, llvm.graph.Constant):
          self.G.add_node(op, attr=(op.type))
          self.G.add_edge(op, v, attr=('data'))
        elif isinstance(op,
                        llvm.graph.Instruction):
          self.G.add_node((v, op), attr=(op.type))
          self.G.add_edge(op, (v, op),
                          attr=('data'))
          self.G.add_edge((v, op), v,
                          attr=('data'))
```

Listing 2: Visitor used to construct LLVM IR graph representations. Overlay colors correspond to edge colors in Figure 3.

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

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