PROGRAML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations Appendices

A. Data Flow Definitions

This section provides the definitions of the five analysis tasks used in this paper to evaluate the representational power of deep learning over programs. We chose a diverse set of analysis tasks to capture a mixture of both forward and backward analyses, and control-, data-, and procedure-sensitive analyses.

(I) REACHABILITY: Reachable Instructions Control reachability is a fundamental compiler analysis which determines the set of points in a program that can be reached from a particular starting point. Given succ(n), which returns the control successors of an instruction n, the set of reachable instructions starting at root n can be found using forward analysis:

$$\text{Reachable}(n) = \{n\} \bigcup_{p \in \text{succ}(n)} \text{Reachable}(p)$$

(II) DOMINANCE: Instruction Dominance Instruction n dominates statement m if every control-flow path the from the program entry n_0 to m passes through n. Like reachability, this analysis only requires propagation of control-flow, but unlike reachability, the set of dominator instructions are typically constructed through analysis of a program's reverse control-flow graph (Lengauer & Tarjan, 1979; Blazy et al., 2015):

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

Where pred(n) returns the control predecessors of instruction n. We formulate the DOMINANCE problem as: Given a root instruction vertex n, label all vertices m where $n \in \text{Dom}(m)$.

(III) DATADEP: Data Dependencies The data dependencies of a variable v is the set of predecessor instructions that must be evaluated to produce v. Computing data dependencies requires traversing the reverse data-flow graph:

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

Where defs(n) returns the instructions that produce the operands of n.

(IV) LIVENESS Live-out variables A variable v is live-out of statement n if there exists some path from n to a statement that uses v, without redefining it. Given uses(n), which returns the operand variables of n, and defs(n), which returns defined variables, the live-out variables can be computed forwards using:

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

(V) Global Common Subexpressions The identification of common subexpressions is an important analysis for optimization. For compiler IRs we define a subexpression as an instruction 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 an instruction (which forms part of a subexpression), label any other instructions in the program which compute the same subexpression. This is an inter-procedural analysis, though operands must obey their scope. Common subexpressions are typically identified using available expression analysis:

$$\text{Avail}(n) = \text{uses}(n) \cup \left(\bigcap_{p \in \text{pred}(n)} \text{Avail}(p) \right) - \text{defs(n)}$$

Where uses(n) return the expressions used by instruction n, and defs(n) returns the expressions defined by n.

B. DEEPDATAFLOW Dataset

The DEEPDATAFLOW dataset comprises: 461k LLVM-IR files assembled from a range of sources, PROGRAML representations of each of the IRs, and 15.4M sets of labeled graphs for the five data flow analyses described in the previous section, totaling 8.5B classification labels. The dataset is publicly available (Cummins, 2020).

Programs We assembled a 256M-line corpus of real-world LLVM-IRs from a variety of sources, summarized in Table 1. We selected popular open source software projects that cover a diverse range of domains and disciplines, augmented by uncategorized code mined from popular GitHub projects using the methodology described by Cummins et al. (2017b). Our corpus comprises five source languages (C, C++, Fortran, OpenCL, and Swift) covering a range of domains 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.

PROGRAML Graphs We implemented PROGRAML construction as an `llvm::ModulePass` using LLVM version 10.0.0 and generated a graph representation of each of the LLVM-IRs. PROGRAML construction takes an average of 10.72ms per file. Our corpus of unlabeled graphs totals 268M vertices and 485M edges, with an average of 581 vertices and 1,051 edges per graph. The maximum edge position is 355 (a large `switch` statement found in a TensorFlow compute kernel).

Data Flow Labels We produced labeled graph instances from the unlabeled corpus by computing ground truth labels for each of the analysis tasks described in Section A using a traditional analysis implementation. For each of the five tasks, and for every unlabeled graph in the corpus, we produce n labeled graphs by selecting unique source vertices $v_0 \in V$, where n is proportional to the size of the graph:

$$n = \min\left(\left\lceil \frac{|V|}{10} \right\rceil, 10\right)$$

Each example in the dataset consists of an input graph in which the source vertex is indicated using the *vertex selector*, and an output graph with the ground truth labels used for training or for evaluating the accuracy of model predictions. For every example we produce, we also record the number of steps that the iterative analysis required to compute the labels. We use this value to produce subsets of the dataset to test problems of different sizes, shown in Table 2.

We divided the datasets randomly using a 3:1:1 ratio for training, validation, and test instances. The same random allocation of instances was used for each of the five tasks. Where multiple examples were derived from a single IR, examples derived from the same IR were allocated to the same split.

As binary classification tasks, data flow analyses display strong class imbalances as only a small fraction of a program graph is typically relevant to computing the result set of an analysis. On the DDF test sets, an accuracy of 86.92% can be achieved by always predicting the negative class. For this reason we report only binary precision, recall, and F1 scores with respect to the positive class when reporting model performance on DEEPDATAFLOW tasks.

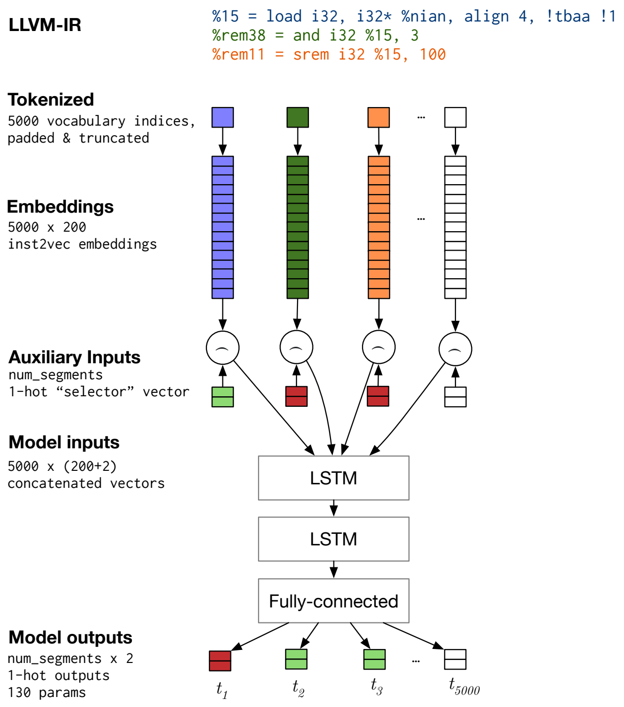

Figure 1: Extending inst2vec (Ben-Nun et al., 2018) to perform per-instruction classification of LLVM-IR. The $\frown$ operator denotes vector concatenation.

C. Data Flow Experiments: Supplementary Details

This section provides additional details for the experiments presented in Section 5.

C.1. Models

(I) Sequential Model The inst2vec model consists of 619,650 trainable parameters in the configuration outlined in Figure 1. The model, implemented in TensorFlow, uses the same parameters as in the original work: $2\times$ 64 dimensional LSTM layers followed by a 64 dimensional dense layer and the final 2 dimensional output layer. Sequences are padded and truncated to 5k tokens and processed in batches of 64.

(II) Graph Models For CDFG and PROGRAML approaches we use the same model architecture. The model, implemented in PyTorch, consists of a customized GGNN with 87,070 trainable parameters. Batches are implemented as disconnected graphs that are constructed to enable efficient processing of graphs of differing size in parallel without padding. We use a combined batch size of $10,000$ vertices. If a single graph contains more than $10,000$ vertices, it is processed on its own.

	Language	Domain	IR files	IR lines
BLAS 3.8.0	Fortran	Scientific Computing	300	345,613
GitHub	C	Various	38,109	74,230,264
	OpenCL		5,224	9,772,858
	Swift		4,386	4,586,161
Linux 4.19	C	Operating Systems	13,418	41,904,310
NPB (Bailey et al., 1991)	C	Benchmarks	122	255,626
Cummins et al. (2017a)	OpenCL	Benchmarks	256	149,779
OpenCV 3.4.0	C++	Computer Vision	432	2,275,466
POJ-104 (Mou et al., 2016)	C++	Standard Algorithms	397,032	104,762,024
Tensorflow (Abadi et al., 2016)	C++	Machine learning	1,903	18,152,361
Total			461,182	256,434,462

Table 1: The DeepDataFlow LLVM-IR corpus.

	DDF-30	DDF-60	DDF-200	DDF
Max. data flow step count	30	60	200	28,727
#. classification labels	6,038,709,880	6,758,353,737	7,638,510,145	8,623,030,254
#. graphs (3:1:1 train/val/test)	10,951,533	12,354,299	13,872,294	15,359,619
Ratio of full test set	71.3%	80.4%	90.3%	100%

Table 2: Characterization of DeepDataFlow subsets.

C.2. Experimental Setup

Training Details and Parameters All models were trained in an end-to-end fashion with the Adam optimizer (Kingma & Ba, 2015) using the default configuration and a learning rate of $1 \cdot 10^{-3}$ for the LSTMs and $2.5 \cdot 10^{-4}$ for the GGNNs. We trained the models on 1M training graphs, evaluating on a fixed 10k validation set at 10k intervals for the first 50k training graphs, and at 100k intervals thereafter. The checkpoint with the greatest validation F_1 score is used for testing.

Runtimes All experiments were conducted on shared machines equipped with an NVIDIA GTX 1080 GPU, 32GB of RAM, mechanical hard drives, and server-grade Intel Xeon processors. Figure 3 provides measurements of the average runtimes of each approach across the five DDF-30 tasks. In our implementation, we find training and testing to be I/O bound as programs are processed faster than loading many small files from disk. In particular, CDFG performance suffers relative to ProGraML as the conversion from ProGraML to CDFG representations is performed on-demand. For validation, inputs are loaded once into system memory and re-used, so the measured time provides a more accurate estimate of processing requirements.

D. Downstream Tasks: Supplementary Details

This section provides additional details for the experiments present in Section 6.

D.1. Heterogeneous Compute Device Mapping

Datasets The OpenCL DevMap dataset comprises 256 OpenCL kernels from two combinations of CPU/GPU pairs. The *AMD* set uses an Intel Core i7-3820 CPU and AMD Tahiti 7970 GPU; the *NVIDIA* set uses an Intel Core i7-3820 CPU and an NVIDIA GTX 970 GPU. Each dataset consists of 680 labeled examples derived from the 256 unique kernels by varying dynamic inputs.

Models We compare ProGraML with four approaches: First, with a static baseline that selects the most-frequently optimal device for each dataset (CPU for *AMD*, GPU for *NVIDIA*). Second, with DeepTune (Cummins et al., 2017a), 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, which adapts DeepTune to using tokenized sequences of LLVM-IR as input instead of OpenCL tokens. Finally, we compare against the state-of-the-art approach inst2vec (Ben-Nun et al., 2018), 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. ProGraML itself uses the GGNN adaptation as described in the paper. We adapted the readout head to produce a single classification label for each graph, rather than per-vertex classifications, by aggregating over the final iterated vertex states. We also included the available auxiliary input features of the DevMap dataset. The auxiliary features are concatenated to the features extracted by the GGNN before classification following the methodology of Cummins et al. (2017a).

	Train time	Test time	Train time/graph	Val time/graph	Test time/graph
inst2vec	10h52m	1h33m	45ms	3ms	36ms
CDFG	13h14m	3h27m	64ms	1ms	62ms
PROGRAML	7h21m	1h39m	26ms	3ms	24ms

Table 3: Average training and inference times on DDF-30 tasks

The experimental results in this section come from an earlier development iteration of PROGRAML which deviates from the method described in the main paper in the way in which it produces initial vertex embeddings. Instead of deriving a textual representation of instructions and data types to produce a vocabulary, the vocabulary used for the DEVMAP experiment is that of inst2vec (Ben-Nun et al., 2018), where variables and constants are all represented by a single additional embedding vector. The poor vocabulary coverage achieved by using inst2vec motivated us to provide the improved vocabulary implementation that we describe in the main paper (see Table 1).

Training Details and Parameters All neural networks are regularized with dropout (Hinton et al., 2012) for generalization and Batch Normalization (Ioffe & Szegedy, 2015) in order to be uniformly applicable to vastly different scales of auxiliary input features. 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, setting aside $1/10$th of the training data to use for validation. We trained each model for 300 epochs and selected the epoch with the greatest validation accuracy for testing. Baseline models were trained with hyperparameters from the original works. For the PROGRAML results we used 6 layers in the GGNN 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 us to use the pre-trained inst2vec statement embeddings (Ben-Nun et al., 2018) 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 256 unique programs. PROGRAML was trained with the Adam optimizer with default parameters, a learning rate of 10^{-3} and a batch size of 18,000 nodes (resulting in ca. 12000 iteration steps of the optimizer). For the PROGRAML result, we repeat the automated sweep for all hyperparameter configurations and picked the configuration with the best average validation performance. Performance on the unseen tenth of the data is reported.

D.2. Algorithm Classification

Dataset We use the POJ-104 dataset (Mou et al., 2016). It contains implementations of 104 different algorithms that were submitted to a judge system. All programs were written by students in higher education. The dataset has around 500 samples per algorithm. We compile them with different combinations of optimization flags to generate a dataset of overall 240k samples, as in Ben-Nun et al. (2018). Approximately 10,000 files are held out each as a development and test set.

Models We compare with tree-based convolutional neural networks (TBCNN) (Mou et al., 2016) and inst2vec (Ben-Nun et al., 2018). We used author-provided parameters for the baseline models. For PROGRAML we used 4 layers in the GGNN corresponding to 8 timesteps. To further test the expressive power of the graph-based representation against the tree-based (TBCNN) and sequential (inst2vec) prior work, we additionally compare against graph-based baselines based on XFG (Ben-Nun et al., 2018).

To better understand the qualitative aspects of replacing a graph-based representation that captures program semantics like Contextual Flow Graphs (XFG) (Ben-Nun et al., 2018) with the more complete PROGRAML representation, we adapted a GGNN (Li et al., 2015) to directly predict algorithm classes from XFG representations of the programs. In contrast to this, Ben-Nun et al. (2018) 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 structure-less sequential approach in the original work (inst2vec).

Training Details and Parameters All models were trained with the AdamW (Loshchilov & Hutter, 2019) optimizer with learning rate $2.5 \cdot 10^{-4}, \beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ for 80 epochs. Dropout regularization is employed on the graph states with a rate of 0.2.

References

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

Bailey, D. H., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS Parallel Benchmarks. *IJHPCA*, 5(3), 1991.

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

Blazy, S., Demange, D., and Pichardie, D. Validating Dominator Trees for a Fast, Verified Dominance Test. In *ITP*, 2015.

Cummins, C. DeepDataFlow. Zenodo, June 2020. URL `https://doi.org/10.5281/zenodo.4247595`.

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

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

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

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

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

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

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

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *ICLR*, 2019.

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