Sequentializing Compiler-Based Graph Representations of Code for Machine Learning

Group 28 - Wenfei Tang, Fangyuan Yang, Cooper Stevens, and Samin Riasat

***Abstract*—In recent years, much research has been done into analyzing source code to determine compiler optimizations to improve runtime on a given hardware. Tasks for compiler optimizations for kernels include CPU/GPU mapping and thread coarsening. [2] Given the resemblance of source code to natural language, a common approach for learning a decision maker for these optimizations given source code includes natural language processing approaches like LSTM-based models. Given the popularity and depth of research into such approaches, these have proven to be quite successful, as many methods and improvements have been proposed for such models.**

However, one can notice distinct differences between natural language and source code (for example, a line of code occurring after another does not necessarily guarantee the same temporal relationship in execution, unlike natural language). As such, researchers have more recently began looking into other forms of representation for source code that may be more conducive to learning a decision maker. In particular, many compiler-based graph representations have been shown to have some merit in recent studies. [2], [4] In this exploration, we look to see if we can improve overall performance by combining the benefit of the compiler-based graph representations (that encode application-specific knowledge) with the benefit of the well-researched, high-performance LSTM-based models by sequentializing the graph representations.

I. Introduction

In recent years, much research has been done into analyzing source code to determine compiler optimizations to improve runtime on a given hardware. Tasks for compiler optimizations for kernels include CPU/GPU mapping and thread coarsening. [2] Given the resemblance of source code to natural language, a common approach for learning a decision maker for these optimizations given source code includes natural language processing approaches like LSTM-based models. Given the popularity and depth of research into such approaches, these have proven to be quite successful, as many methods and improvements have been proposed for such models.

However, one can notice distinct differences between natural language and source code (for example, a line of code occurring after another does not necessarily guarantee the same temporal relationship in execution, unlike natural language). As such, researchers have more recently began looking into other forms of representation for source code that may be more conducive to learning a decision maker. In particular, many compiler-based graph representations have been shown to have some merit in recent studies. [2], [4] In this exploration, we look to see if we can improve overall performance by combining the benefit of the compiler-based graph representations (that encode application-specific knowledge) with the benefit of the well-researched, high-performance LSTM-based models by sequentializing the graph representations.

II. Related Works

A. Deep Learning of Optimization Heuristics

Early models for predicting compiler optimizations rely on manually designed IR-level features. To extract features without expert domain knowledge and trial and error, several deep learning models for building heuristics have been proposed: Cummins et al. [5] developed DeepTune, a deep neural network that predicts compiler-internal optimization heuristics based on source code. In DeepTune, an OpenCL program is first generated as tokens and mapped to a sequence of embedding vectors and then processed by a sequential LSTM recurrent neural network model. With some optionally program-level features, the final state of LSTM is fed into a fully connected neural network to generate a program-level classification.

B. Graph Representation

CDFG Brauckmann et al. [2] proposed an LLVM-based control and dataflow graph enriched with calls and memory nodes. In CDFG, the nodes are labeled with LLVM IR instructions. 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.

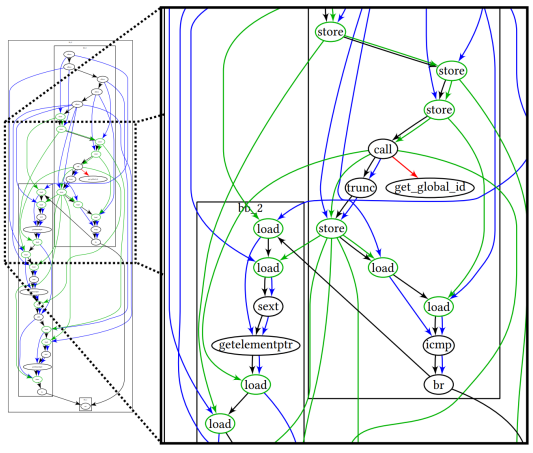

Fig. 1. An example of CDFG representation. The control flow is depicted by black arrows, the data flow by blue ones, memory dependency by green ones, and the red arrow represent an external function call.

PROGRAML Cummins et al. [4] enriched the CDFG in terms of program representation: CDFGs ignore the data elements of programs, and only instruction opcodes are used for vertex embeddings. The latent features are thus invariant to instruction operands, their order, data types, and instruction modifiers.

PROGRAML represents programs as directed graphs where statements, identifiers, and immediate values are vertices. First, it constructs a full-flow graph from an IR by inserting a graph vertex for each instruction and control flow edges between them. Second, it introduces additional graph vertices for constant values and variables and adds dataflow edges to capture relations from constants and variables to the instructions' operands. Finally, call sites are extracted, and call edges are inserted from call sites to function entry statements, and from function exit vertices to call sites.

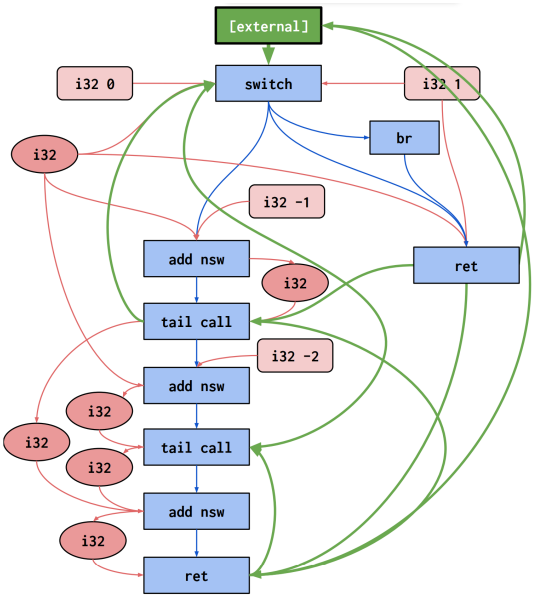

Fig. 2. An example of a PROGRAML representation.

C. Graph-to-Sequence Model

The Sequence-to-Sequence learning technique and its numerous variants achieve excellent performance on many tasks. However, existing SEQ2SEQ models face a significant challenge in achieving accurate conversion from graph form to the appropriate sequence given many machine learning tasks have inputs naturally represented as graphs. Xu et al. [6] introduced a novel general end-to-end graph-to-sequence neural encoder-decoder model that maps an input graph to a sequence of vectors and uses an attention-based LSTM method to decode the target sequence from these vectors. The attention mechanism they proposed aligns node embeddings and the decoding sequence to better cope with large graphs, which is helpful for large-scale graph representation like PROGRAML.

III. METHOD

A. Motivation

Prior work [2] in compiler-based graph representations proposes new machine learning representations to faithfully represent the code semantic structure, such as the LLVM-based control-and dataflow graph enriched with calls and memory nodes [2] and the dataflow-enriched AST [2]. Through encoding more information than a sequence of code or a sequence of IR representations, these two representations clearly expose the semantics of code in a structural way, while discarding unnecessary information such as the over-constrained ordering in the sequential representation of the code.

Experiments have been done in [2] using the two representations proposed with a GNN-based machine learning architecture in this paper. While the results for the AST representation failed to outperform state-of-the-art sequential models(i.e. DEEPTUNE [5]), the CDFG representation shows promise in that it was able to outperform the state-of-the-art. Given the demonstrated improvements for the CDFG representation, we look to take advantage of this seemingly superior representation of code.

B. Our Work

It is not surprising to see that the DEEPTUNE [5] model could have such an impressive performance even though the input representation of this model is simply a series of C tokens; tremendous study and research in the NLP community in the last few years has brought many powerful deep learning models for sequential representations, from the commonly used LSTM to recently trending transformers. The embeddings learned through these deep learning models might reveal the relationships between the IR instructions in a way that we are yet to understand. Given the depth of research and proven high performance of these sequential models, we posit that we may be able to make the most of both sides and: high-performance deep learning models for sequential representations and the information-rich compiler-based graph representations. Our investigation looks to see if utilizing both of these can increase overall performance. The general architecture of our work is shown in Figure 3.

Therefore, we propose to build an architecture that can combine both graph representations and deep learning models for sequential representations.

C. Architecture Design

The high-level idea of this architecture is that we will first generate some sequential representations from the compiler-based graph representations and feed the transformed sequential representations into a deep learning model which takes in sequential representations. Thus, the prediction task can be performed with the sequential model. We use an LSTM-based neural network.

For the module which transforms graph representations, we base our design on previous work on graph to sequence learning tasks. The GRAPH2SEQ [6] model that we eventually use is an end-to-end graph-to-sequence neural encoder-decoder

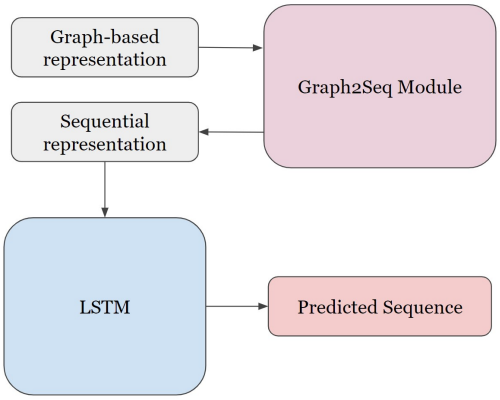

Fig. 3. General Architecture for the proposed pipeline.

model that maps an input graph to a sequence of vectors. Though this model is originally designed for path-finding problems and Natural Language Generation tasks, we were able to incorporate their model into our pipeline for the compiler-based task.

After we obtained the sequential representations from the GRAPH2SEQ module, we feed the representations directly into LSTM. Though a, perhaps, better design choice could be made here by using a transformer-based sequential deep learning model, our computing resources are limited (by Google Colaboratory) and we cannot afford to train a transformer-based model. Thus, the LSTM model is chosen, and the learning target (i.e. the ground truth output sequence) will be formatted based on our evaluation task. In the device mapping evaluation task that we talk about in the next section, we define the output sequence as a value indicating whether to do the mapping or not.

D. Graph Representations and Data Preparation

The graph-representations we use for the input code is generated based on the enriched LLVM-based control-and dataflow graph in previous compiler-based graph representation work [2].

Given an enriched CDFG graph, we assign different attributes to different nodes based on the LLVM parsing results. This is similar to the text feature for each node in a graph-based text representation input, in that every word in the input text has a unique meaning. In our problem setting, we consider different types of nodes having different "meanings" in the compiler-based representation.

When building the adjacency list for each node, one node can appear more than once in another node's adjacency list, if the two nodes are found to be connected by the call edges or the memory edges defined in the paper [2].

The final enriched CDFG-based representation will consist of the nodes, edges, adjacency lists, and attributes for each node. Then, this representation will be passed to the GRAPH2SEQ module, outputting a sequential representation through learning an embedding for each node.

IV. EVALUATION

A. CPU/GPU Mapping Task

One of the common downstream tasks used to evaluate compiler-based representations is to solve the problem of CPU/GPU mapping on a heterogeneous system. In this problem, we are given a heterogeneous system and the source code of a kernel, and are tasked with picking either the CPU or the GPU to execute the kernel on. We consider the decision to be correct if we chose the hardware that would result in a faster execution. The heterogeneous system that we use for our tests has an AMD Tahiti 7970 GPU. See 4 for the example code in this device mapping task.

```
__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];
}
```

Fig. 4. An example of an OpenCL kernel code [2]

B. Experimental Setup

We ran several experiments to evaluate our model. Table I shows a comparison of the accuracy of our model for various choices of hyper-parameters. The model parameters that we vary in an attempt to generate better models include embedding dimension, training batch size, training epochs, and layer size. At a high level, embedding dimension and layer size are proportional to model complexity, while training batch size and training epochs determine the model's exposure to the training data.

We considered CDFG and PROGRAML representations as input data to our model. Note that while the original model used 100 embedding dimensions for the NLP task, 10 dimensions were sufficient for the compiler-based representation to obtain relatively good accuracy. Furthermore, for the CDFG representation our model converged rapidly with good performance. In particular, we needed to train for only 100 epochs to obtain a 75% accuracy compared to DEEPTUNE [5], which needed 1000 epochs to obtain a 79% accuracy (cf. Table II).

We can see that for the CDFG representation, the hyper-parameters in the first row gave the best accuracy, and the accuracies in general were comparable. We however faced some challenges while training our model on PROGRAML data. PROGRAML representations are harder to learn because the representation size is much larger. In particular, we were unable to train our model on PROGRAML data with batch size 32 and layer size greater than 1 due to resource limitations.

All of our experiments were run on Google Colaboratory. Each training epoch took about 2 minutes to complete. The training for the fourth experiment ended abruptly at epoch 32 due to hardware restrictions enforced by Google Colaboratory. Notably, this model still reported a non-trivial accuracy.

TABLE I
ACCURACY OF GNN-BASED MODELS FOR DIFFERENT CHOICES OF HYPER-PARAMETERS

Model	Node embedding dimension	Train batch size	Sample layer size	Training epochs	Accuracy on test set
CDFG	10	32	4	100	0.75
CDFG	10	16	4	100	0.74
CDFG	50	32	4	100	0.71
CDFG	10	32	8	32	0.66
CDFG	150	32	4	100	0.70
CDFG	100	32	8	100	0.72
CDFG	100	32	16	100	0.70
PROGRAML	10	32	1	100	0.65
PROGRAML	100	4	4	100	0.67

TABLE II
COMPARISON OF GNN-BASED MODELS WITH LSTM-BASED MODELS [3]

	Model	Accuracy
GNN	CDFG	0.75
	PROGRAML	0.67
LSTM	DEEPTUNE [5]	0.79
	Barchi et al. [1]	0.76

V. CONCLUSION

To conclude, we have seen that serialized representations of compiler-based graph representations of code have yielded acceptable results for the task of mapping a kernel to a CPU/GPU. However, we have also seen that this representation was unable to outperform state-of-the-art models for the same task. We recognize that these results are premature to consider this investigation fully closed, as these results were produced under relatively restricted conditions, such as hardware restrictions from Google Colaboratory, whereas the state-of-the-art results (such as that for DEEPTUNE) were produced under unknown hardware restrictions, and are likely to have had access to more advanced hardware. Therefore, further investigation into the utility of our proposed representation may be worth while in future work, as the results have proven to be promising thus far.

Hence, for future work, one might look to research what improvements can be made to these models by introducing additional model complexity (which would require more advanced hardware than we had access to for this investigation). One way to do this is to sequentialize the graph representations in such a way that it is compatible with more advanced networks like DEEPTUNE. One might also look into other methods of serializing graph representations. For example, one may look into ways of encoding edge information into a serial representation of a PROGRAML graph.

REFERENCES

[1] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva, "Code Mapping in Heterogeneous Platforms Using Deep Learning and LLVM-IR," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317789

[2] 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*, 2020, pp. 201–211.

[3] A. Brauckmann, A. Goens, and J. Castrillon, "Compy-learn: A toolbox for exploring machine learning representations for compilers," in *2020 Forum for Specification and Design Languages (FDL)*, 2020, pp. 1–4.

[4] C. Cummins, Z. Fisches, T. Ben-Nun, T. Hoefler, M. O'Boyle, and H. Leather, "ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations," in *Thirty-eighth International Conference on Machine Learning (ICML)*, 2021.

[5] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-End Deep Learning of Optimization Heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 219–232.

[6] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2Seq: Graph to Sequence Learning with Attention-based Neural Networks," *arXiv preprint arXiv:1804.00823*, 2018.