CogR: Exploiting Program Structures for Machine-Learning Based Runtime Solutions

Hyojin Sung[†,*], Tong Chen[*], Alexandre Eichenberger[*], and Kevin K. O'Brien[*]

[†]Pohang University of Science and Technology
hsung@postech.ac.kr
[*]IBM T. J. Watson Research Center
[chentong,alexe,caomhin]@us.ibm.com

***Abstract*—We propose CogR, a machine-learning based runtime solution, that enables efficient and dynamic resource scheduling and performance optimization for high-level programming interfaces on heterogeneous systems. CogR tightly combines the structural information of programs and fine-grained static and dynamic statistics into sequenced input data. This structural and value-embedded representation of programs enables CogR to accurately model the runtime behaviors of nested loop-based constructs in the high-level parallel programs. The end-to-end CogR system consists of compiler and runtime support for feature collection and input generation, a machine learning model, and a runtime scheduler with online inference and prediction. The system provides 11% higher prediction accuracy than models simulated for prior work and improves kernel performance by 66% compared to the baseline runtime.**

I. Introduction

Modern systems with intra- and inter-accelerator parallelism pose an unprecedented challenge for the entire software stack to provide programmability and code/performance portability while delivering performance. High-level programming interfaces [1] address this challenge with loop-based parallel and offloading constructs but heavily depend on compiler and runtime for efficient code generation and scheduling on heterogeneous systems with varying configurations.

In our effort to explore the machine-learning based approach as an alternative to traditional heuristics for more flexible and efficient scheduling, we made the following key observation: In order to accurately model nested loop-based constructs and predict their performance, it is crucial to explicitly encode structural information in the input data and, furthermore, *embed fine-grained static and dynamic statistics at the precise contextual point in the sequence*. Fine-grained value-embedded representation separates our approach from previous work that used AST or program source codes [3], [8].

Thus, we propose *CogR*, a machine-learning based runtime solution for OpenMP programs on heterogeneous systems. We aim to guide the OpenMP runtime scheduler by predicting whether an OpenMP offloading construct will execute faster on CPU or GPU. The *CogR* system consists of the three main components shown in Figure 1: (1) Feature extractor and input generator in compiler and runtime, (2) machine learning model trained with the collected features and used for online inference, and (3) runtime scheduling interface. Our

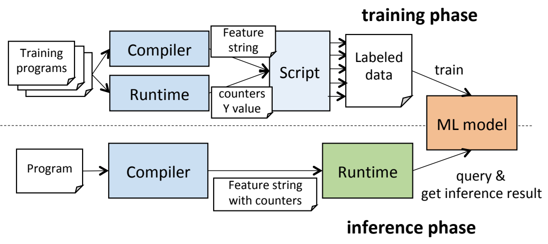

Fig. 1. The Overall Architecture of *CogR*

experimental results show that the *CogR* system provides 11% higher prediction accuracy on average than models trained with structural data with top-level counters (simulated for [3]), structural data only (simulated for [8]) and counters only (simulated for [4]). It also improves the end-to-end performance of our training examples by 66% on average, compared to the baseline that always offloads kernels to GPU.

II. Design and Implementation of CogR

Input Definition: We encode the syntactic structure of a program in our input data and embed fine-grained statistics precisely next to a related syntactic token. Our input data is a sequence of such value-embedded syntactic nodes from traversing the abstract syntax tree (AST) top-down, prefixed by the y value for labeled data during training and followed by padding to produced fixed-length input. The vocabulary for the syntactic tokens includes all OpenMP scoped and standalone directives, non-declaration program statements with side effects, and functions calls. For efficient training, we abstract out lower-level syntactic elements. We embed the following static/dynamic counters next to the token related to them: the number of OpenMP *map* clause and *reduction* clause variables, loop instruction count, and loop iteration count.

Feature Extractor and Input Generator: The *CogR* compiler pass extracts syntactic tokens and compile-time statistics from the AST of a program by visiting each node in an AST for an OpenMP *target* region. The *CogR* runtime profiling interface collects iteration counts for loops in a *target* region from program executions (training) or instrumentation (inference). It also measures program execution time on both CPU and GPU to obtain y values. Iteration counts are keyed by source code location information, e.g., file name/ID and line number, which allows us to merge them with the features from the compiler pass.

Machine Learning Model: We use a variation of recurrent neural networks (RNN) [6], [5], [2] that are capable of structurally correlating tokens apart in input sequence using temporal memory. Our input sequence consists of data of two distinct types, symbolic tokens for syntactic elements and numeric counters, which can have values from widely different ranges. We applied skip-gram model embedding techniques [7] for syntactic tokens and value normalization for numeric counters to generate "learnable" feature vectors based on them. The overall RNN-based neural network is composed of a skip-gram embedding model, an RNN layer, and then a number of fully connected (FC) layers. The final output from the FC layers is the model's decision for CPU or GPU execution.

Runtime Scheduling Interface: We extended the existing OpenMP runtime to communicate with the compiler to get static features, obtain dynamic loop iteration counts, and combine them to generate final input data online. Right before executing an OpenMP *target* region, the runtime performs inference on the trained model and uses the result to schedule the region on CPU or GPU. To reduce the online inference overheads, we overlapped the inference query with program execution and cached inference results to reuse them for later instances of the same *target* region.

III. Experimental Result

We extended Clang 4.0 with full support for OpenMP 4+ and LLVM 7 for the *CogR* compiler pass, and a proprietary OpenMP runtime for profiling, online input generation and inference. For training data, we manually ported sixty benchmarks and microkernels to OpenMP with offloading directives and generated 1,027 examples from them. We ran our experiments on an OpenPower node with POWER 9 processors attached via NVLINK to NVIDIA Volta GPUs. For online inference, we set up a standard Linux socket server that serves an inference request from the *CogR* runtime.

We evaluated the RNN-based model for *CogR* along with three models that simulate prior work (*CountersOnly, StructAndCounters, StructOnly*) for comparison. Embedding dimensions and network parameters including RNN cell type and size, and the number of RNN and FC layers are determined by extensive sensitivity study for each model. The models are implemented in TensorFlow v1.5, and we performed a k-fold validation with k=20 to obtain final accuracy numbers.

Prediction Accuracy: Figure 2 shows that the *CogR* model outperforms the other models by 11% on average with 88% prediction accuracy. *CogR* significantly outperforming *StructOnly* and *StructAndCounters* in all statistics shows that *CogR* excels them in truly modeling program behaviors by not simply adding but tightly integrating fine-grained statistics into structural data. Overall, our evaluation of the models with a different input format confirms our hypothesis that structural and value-embedded data improves learning efficiency compared to non-structural data or structural data with flat summary counters.

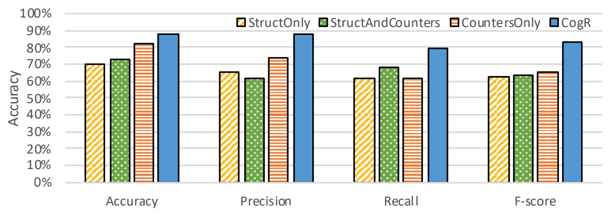

Fig. 2. Model Accuracy

Client Initialization	87 us
Feature String Generation	6 us
Inference and Server Communication	3.9 ms

TABLE I
CogR Runtime Overhead

End-to-End Performance: We measured the end-to-end kernel execution times on CPU or GPU as predicted by *CogR* for all kernels in our training set. Then we calculated the average difference between them and the execution times by the baseline runtime that respects scheduling directions from source codes, which always offloads to GPU if possible. The result showed that *CogR* improved kernel performance by 66% on average compared to the baseline.

CogR Overhead: We found the overhead incurred by the *CogR* runtime (as shown in Table III) to be consistent across the scheduling instances evaluated. The majority of the overhead comes from inference time during online inference. The optimizations described in Section II reduced the inference time from 10ms to 3.9ms by 61% with asynchronous server communication and eliminated the entire trip by using cached results with 91% accuracy against actual inference results.

IV. Conclusion and Future Work

Overall, *CogR* showed its potential as a realistic machine-learning based runtime solution with full compiler and runtime support for input data generation and online inference. Our evaluation showed that the expressiveness of our structural and value-embedded input data enabled more accurate modeling of program behaviors. We plan to extend *CogR* to a broader range of problems and target applications, while further improving the model accuracy and reducing the runtime overheads.

References

[1] O. A. R. Board. The openmp application programming interface. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[2] K. Cho *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[3] C. Cummins *et al.*, "End-to-end deep learning of optimization heuristics," in *PACT 2017*.

[4] H. Eom *et al.*, "Machine learning-based runtime scheduler for mobile offloading framework," in *UCC 2013*.

[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8.

[6] L. C. Jain and L. R. Medsker, *Recurrent Neural Networks: Design and Applications*, 1999.

[7] O. Levy and Y. Goldberg, "Dependency-based word embeddings," in *ACL*, 2014.

[8] L. Mou *et al.*, "Convolutional neural networks over tree structures for programming language processing," in *AAAI 2016*.