
DEEP GRAPH LEARNING FOR PROGRAM ANALYSIS AND SYSTEM OPTIMIZATION

Yao Xiao¹ Guixiang Ma² Nesreen K. Ahmed² Theodore L. Willke² Shahin Nazarian¹ Paul Bogdan¹

ABSTRACT

It has been increasingly challenging for the compilers to cope with the evolving computer architectures. The manually written compiler heuristics are not sufficiently wise to capture the impact of data and hardware related dependencies on performance. However, machine learning offers an opportunity to learn the common patterns in the existing dataset and predict the future outcomes for unseen data. Therefore, rather than relying on expert compiler writers to develop clever heuristics to optimize the code, we can utilize machine learning to optimize a compiler to make the machine run faster. In this work, we represent high level programs as weighted graphs. This enables the proposed framework to efficiently analyze the structural information flow of software programs and determine their parallelization. Furthermore, the proposed framework utilizes graph autoencoders to learn how to partition the graph into computational kernels, and exploits graph neural networks to predict the correct assignment to a processor type. In the evaluation, we validate the PGL framework and demonstrate a maximum speedup of 6.22x when compared to the thread-based execution and 1.91x higher compared to the state-of-the-art PROGRAML.

1 INTRODUCTION

The recent technological advances have significantly contributed to a rapid increase in algorithmic complexity of various applications, from digital signal processing to autonomous aerial, ground and underwater systems (Krishnan et al., 2019). In order to control and manage this increased algorithmic complexity, heterogeneous computing systems require intelligent, flexible and highly efficient programming strategies to provide high performance while minimizing energy costs (Xiao et al., 2019). However, the current monolithic programming models and task mapping to computing engines do not fully exploit the recent architectural innovations and can exacerbate the load imbalance and communication inefficiencies (Xiao et al., 2017).

In order to fully utilize the capabilities of hardware platforms, the compilation of parallel programs depends on expert programmers to use heuristics to decide how many threads to spawn and how to schedule them onto heterogeneous computing systems (Cummins et al., 2017). Due to workload imbalance, synchronization overhead, and resource sharing contention, the overall performance may lead to sub-optimal executions. To address these issues, re-

searchers (Cummins et al., 2017; 2020; Grewe et al., 2013) propose and solve the device mapping problem – given a kernel, how to predict the correct processor, i.e., CPU or GPU, to provide better performance – by developing machine learning approaches to outperform the inefficient heuristics. However, as applications become more diverse and complex, it is inefficient to map them only onto one type of processors. For example, autonomous car driving distributes the visualization and recognition tasks, full of *for* loops, onto cores in GPUs to provide higher parallelization. At the same time, sequential decisions based on *if-else* statements require CPUs to provide the fast execution on a single critical thread. There is a tradeoff between GPUs and CPUs. GPUs provide a higher number of compute engines for parallel computing whereas CPUs have higher frequencies compared to GPUs, leading to a faster execution of sequential threads.

To combine the benefits of both CPUs and GPUs, as opposed to the traditional device mapping problem, we formulate a new problem to be considered within the high performance computing and machine learning contexts:

Given a complex software application, the goal is to learn a mapping function that predicts which code segments would run best on a specific hardware device in heterogeneous hardware platforms.

¹Department of Electrical and Computer Engineering, University of Southern California ²Intel Labs, Intel, USA. Correspondence to: Yao Xiao <xiaoyao@usc.edu>.

Computations in programs can be considered as a graph

where each node represents a compute instruction and each edge represents an information flow from one instruction to another. This graph representation of programs enables us to model the dynamic dependency structures of software programs and helps analyze program characteristics and automatically compile programs in heterogeneous platforms.

Therefore, to solve the challenging optimization problem, we propose a *unified end-to-end programmable graph representation learning* (PGL) framework capable of mining the complexity of high level programs down to the universal intermediate representation, extract the specific computational patterns and predict which code segments run best on a specific core in heterogeneous hardware platforms. We first model each application as a dynamic dataflow graph where nodes represent low level virtual machine (LLVM) intermediate representation (IR) instructions and edges represent control, data, and memory dependencies. Next, we build a graph autoencoder to partition the graph from the complex application into several kernels and a graph neural network (GNN) to predict the correct label for each kernel.

We evaluate the proposed PGL framework on a heterogeneous platform consisting of 32 CPUs and 32 GPUs. The GNN is first trained with C styled kernels converted from OpenCL from seven benchmark suites to learn the weights of GNNs. Next, we integrate the trained GNN model with the GAE into the framework and test new incoming applications. Experimental results demonstrate a maximum speedup of 6.22x when compared to the thread-based execution and 1.91x higher compared to the state-of-the-art techniques.

Contributions. Our main contributions are as follows:

- We formulate a new challenging system optimization problem to be considered in the areas of machine learning and computing systems: Given a software program, the goal is to learn a mapping function that predicts which code segment should run on which hardware device in a heterogeneous computing system.
- We propose a unified end-to-end programmable graph representation learning framework (PGL) that automatically maps the computations of complex software applications to the appropriate hardware device in heterogeneous hardware platforms.
- We conduct extensive experiments and baseline comparisons to validate the PGL framework which achieves an application performance improvement up to 6.22x when compared to the thread-based execution and 1.91x compared to the state-of-the-art techniques.

2 RELATED WORK

We summarize the related work into two areas: (1) deep learning models in compiler optimization, and (2) graph

representation learning for code representation.

Deep Learning in Compiler Optimization. Heuristics used in compilers require expert knowledge to optimize programs on heterogeneous systems and often lead to sub-optimal performance due to synchronization overhead and resource management. Machine learning techniques, in particular deep learning methods, are being applied during the optimization phase to generate efficient machine code (Ashouri et al., 2018; Li et al., 2020; Haj-Ali et al., 2019). The recent work in (Zhou et al., 2020) proposed an end-to-end deep reinforcement learning method for ML compiler graph optimizations where the learned policies are general to new graphs and transferable to different tasks. (Haj-Ali et al., 2020) proposed an end-to-end framework utilizing deep reinforcement learning (RL) for handling loop vectorization. In addition, machine learning techniques are also used to optimize the execution time of tensor computation graphs (Jinnai et al., 2019) as well as deep neural networks in TASO (Jia et al., 2019) and SOAP (Jia et al., 2018).

Graph Representation Learning for Code Representation. While many prior works have employed machine learning methods from natural language processing to represent programs as sequence of lexical tokens (Nguyen et al., 2018; Cummins et al., 2017), recently there emerged a number of graph-based machine learning works that aims to capture the structure of programs along with the syntactic and semantic information in the graph representation (Alon et al., 2019; Ben-Nun et al., 2018; Brauckmann et al., 2020). It has been observed that the graph-based representation learning strategies tend to have superior learning capabilities on the programs for many code analysis tasks, such as code similarity learning (Li et al., 2019), program classification (Mou et al., 2016), etc. For instance, (Brauckmann et al., 2020) uses abstract syntax trees (ASTs) and control-data flow graphs (CDFGs) independently to represent programs and apply GNNs for learning predictive compiler tasks on these graphs, which outperforms the recurrent neural networks (RNNs) on the token sequence representation of the programs. (Cummins et al., 2020) models the program’s control, data and call dependencies as a graph, and applies a GNN to learn representations from the graph for both node-level and graph-level tasks including compiler analysis, program classification and device mapping.

However, compared to previous frameworks, we propose the new heterogeneous device mapping problem that states that given an application, the goal is to learn a mapping function that predicts which code segments benefit most on a specific core in heterogeneous hardware platform. We propose a *unified end-to-end programmable graph representation learning* (PGL) framework to solve the challenging task of heterogeneous device mapping to learn to automatically map

the computations of complex software applications to the appropriate hardware device in heterogeneous platforms.

3 PROGRAMMABLE GRAPH LEARNING (PGL)

In this section, we describe the proposed PGL framework, which consists of three steps. The descriptions of these steps are detailed in the next sections. Section 3.1 discusses the general approach to transform an application into a dynamic dataflow graph. Then, sections 3.2.1 and 3.2.2 discuss the GAE graph partitioning and GNN heterogeneous device mapping prediction, respectively.

3.1 Input Program Modelling

Recently, various graph representations were proposed to represent and capture the latent information flow in a program (e.g., abstract syntax tree (AST) (Alon et al., 2019), contextual flow graph (XFG) (Ben-Nun et al., 2018), and control and data flow graph (CDFG) (Brauckmann et al., 2020)). These graph representations allow the compiler to analyze the effectiveness and correctness of programs, as well as enable parallel programming via graph partitioning in high performance computing (Xiao et al., 2017). However, these statically compiled graphs have several limitations. First, memory dependencies are difficult to be identified. If not handled properly, this can exacerbate the data communication overhead and reduce the application performance. Second, the number of iterations in *for* and *while* loops cannot be statically determined. Therefore, in order to overcome these drawbacks, we use information generated from static compiler analysis and dynamic compilation to model the information flow in programs as a dynamic dataflow graph. Next, we propose the following representation.

Definition 3.1 (DYNAMIC DATAFLOW GRAPH). A dynamic dataflow graph is a weighted directed acyclic graph $G = (V, E, W)$, where each node v , associated with an attribute va indicating the type of the node (e.g., add, sub, store, or load), $(v, va) \in V$ represents an LLVM IR instruction; each edge e , associated with an attribute ea indicating the type of dependencies (e.g., control, data, or memory), $(e, ea) \in E$ represents a dependency between two instructions; a weight $w \in W$ on each edge e represents the amount of data communication between two instructions and the time to execute the instruction. It allows us to quantify the cost of data movement in the memory hierarchy with L1, L2, and L3 caches.

To construct these dynamic dataflow graphs, we first collect the representative dynamic trace generated from executing a program. This trace contains a sequence of LLVM IR instructions to be executed. Then, for each instruction, we

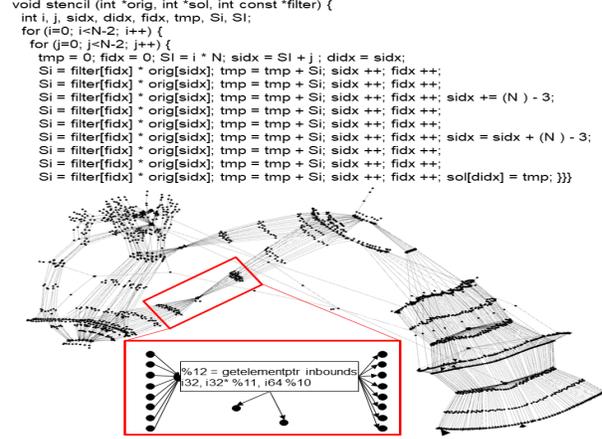


Figure 1. An example of a standard two-dimensional nine point stencil calculation and its corresponding graph representation. By adopting this graph representation, we can see that some patterns are recurring due to for loops used in the code.

check if data, control, and memory dependencies exist and insert directed edges to construct the graph. In order to correctly represent a program, we maintain the control flow graph of a program. An execution path with respect to a seed is a series of basic blocks traversed during the execution. If a certain path is exerted, we color the corresponding nodes in the control flow graph. Once the graph is fully colored, we stop the execution and collect the dynamic trace.

Figure 1 shows the graph representation of a simple program. Note that a node, as indicated in the red box, is an LLVM IR instruction, not an operand or a high level language (e.g., C/C++, Java) statement. Different from AST, XFG, and CDFGs, this specific graph representation reveals the hidden program information flows. One most recurring pattern is the cone structure due to the LLVM IR "getelementptr" generated from the pointers in *for* loops to distribute data to different iterations.

3.2 Graph Representation Learning

Once we extracted the initial node features from the dynamic dataflow graph, we design a deep graph representation learning module with GNNs (Wu et al., 2020) for the graph partition and device mapping prediction problem. Specifically, we propose to use a graph autoencoder (GAE) for partitioning the graph into kernels and a GNN such as graph convolutional network (GCN) for the label prediction.

3.2.1 GAE-based Graph Partitioning

Graph auto-encoders (GAEs) (Zhou et al., 2018) are a category of GNNs that aims at representing nodes into low-dimensional vectors in an unsupervised training fashion. They are different from other GNNs that are typically used for supervised or semi-supervised learning tasks. In our

framework, the goal of the graph partitioning stage is to obtain a good partition for each LLVM graph based on a learned representation that captures the intrinsic structural information of the graph, such that the subgraphs preserve the inherent characteristics of the data, control and memory dependencies in the LLVM graph. To this end, we propose a graph partitioning strategy based on the GAE and spectral clustering (Von Luxburg, 2007) for our task. Given the graph $G = (V, E)$ with an adjacency matrix \mathbf{A} and node features in an $N \times D$ matrix \mathbf{X} , we apply the graph auto-encoder (GAE) model introduced in (Kipf & Welling, 2016) with two graph convolutional layers. We calculate embeddings \mathbf{Z} and the reconstructed adjacency matrix as follows:

$$\hat{\mathbf{A}} = \sigma(\mathbf{Z}\mathbf{Z}^\top), \text{ with } \mathbf{Z} = GCN(\mathbf{X}, \mathbf{A}) \quad (1)$$

After we obtain the node embeddings via GAE, we use spectral clustering (Von Luxburg, 2007) on the node embeddings for the graph partitioning. The overall workflow of this stage is as follows: We first perform the GAE with two graph convolutional layers to learn the latent embedding \mathbf{Z} . Next, we maintain an inner product decoder $\hat{\mathbf{A}} = \mathbf{Z}\mathbf{Z}^\top$ to learn the pairwise distance between nodes. We then perform spectral clustering after calculating the symmetric and non-negative distance matrix $\mathbf{D} = \frac{1}{2}(|\hat{\mathbf{A}}| + |\hat{\mathbf{A}}|^\top)$.

3.2.2 GNN-based Device Mapping Prediction

Once the graph is partitioned into different clusters/kernels, next for each kernel, we use a GNN to predict the correct platform to execute the kernel by updating the node vectors iteratively in a similar fashion to the message passing. Note that our proposed PGL is a general framework that can leverage various GNN models for the device mapping prediction stage, whereas in this paper, we adopt three different variants of the GNN models: GCN, graph attention network (GAT) and gated graph neural network (GGNN), respectively, for this task. We also empirically investigate the comparative effectiveness of these GNN strategies in representation learning on the partitioned LLVM graphs for the graph classification task in heterogeneous device mapping. Once the feature embedding is learned from the GNN models, we use two fully connected feed-forward neural network layers to predict the correct label for each kernel.

4 EXPERIMENTS

Setup. Our framework discussed in the previous section consists of two components: a GAE and a GNN. Unsupervised learning model GAE is used to partition the new complicated program into several clusters / kernels to be mapped onto heterogeneous systems. Supervised learning model GNN predicts the correct label for each kernel. In the implementation, we use kernels written in OpenCL (Cum-

Table 1. Configuration parameters

CPU	Cores	32 cores, 16 MSHRs
	Clock frequency	2.4 GHz
	L1 private cache	64KB, 4-way associative 32-byte blocks
	L2 shared cache	256KB, distributed
	Memory	4 GB, 8 GB/s bandwidth
GPU	Core	32
	Clock frequency	575 MHz
	Memory capacity	768 MB
	Memory bandwidth	86.4 GB/s
Network	Topology	Mesh
	Routing algorithm	XY routing
	Flow control	Virtual channel flit-based

Table 2. Applications and descriptions. We use the following eight benchmarks to validate the benefits of the PGL framework whereas we use the dataset (Cummins et al., 2017) to train the graph neural network in the framework.

Application	Description	Input Size
dijkstra	Find the shortest path	100 nodes
fft	Fast Fourier transform	vector of size 4096
k-means	K cluster partitioning	256 2D tuples
mandel	Calculate Mandelbrot set	4092 points
md	Molecular dynamics	1024 particles
nn	Neural network	5 hidden FC layers
neuron	ReLU neurons	1024 neurons
cnn	Conv. neural network	conv-pool-FC

mins et al., 2017) as training and testing data with 5-fold cross validation. The ground-truth labels are either CPU or GPU for the kernels. Once the GNN is trained, we use new applications, whose statistics are summarized in Table 3, as input graphs to the framework to find kernels for CPUs and GPUs. Table 1 lists the configuration parameters of the heterogeneous system used in this section. In the evaluation, we first report the total accuracy of the GNN on the testing data and then we use programs shown in Table 2 and map kernels onto the heterogeneous system to measure the performance improvement over the state-of-the-art methodologies. We report the normalized speedup in terms of the application performance as the thread-based execution time (slowest) divided by the execution time for each approach.

Datasets. We start by using the 256 heterogeneous device mapping OpenCL kernels in (Cummins et al., 2017) for training and validation of GNNs. These kernels are labelled with CPU vs. GPU and are collected from seven benchmark suites. We then manually convert these kernels to C code. We use the NVIDIA set with an Intel Core i7-3820 CPU and an NVIDIA GTX 970 GPU. Furthermore, we use standard application benchmarks (see Table 2 for details) to validate the overall proposed PGL framework in comparison with baselines.

Baseline Comparisons. When comparing the accuracy of the prediction results from GNN models, we use the

Table 3. Graph statistics.

Program	No. Nodes	No. Edges	Avg Path Length
dijkstra	502,897	588,046	17.36
fft	456,183	572,053	15.54
k-means	705,184	839,125	22.18
mandel	235,051	260,042	11.67
md	1,799,353	2,361,213	34.29
nn	227,766	286,714	19.22
neuron	987,184	1174,843	52.75
cnn	361,464	520,596	13.34

following baselines: (1) GCN; (2) GAT; and (3) GGNN. We compare the PGL framework with PROGRAML (Cummins et al., 2020), NCC (Ben-Nun et al., 2018), and DeepTune (Cummins et al., 2017), state-of-the-art techniques to represent programs as graphs, to validate the effectiveness of our graph representation. To quantify the benefits of graph partitioning, we compare the PGL framework with the following baselines in terms of the application performance: (1) K-means clustering connected with GCNs (KM+GCN); (2) hierarchical divisive clustering where all observations start in one cluster, and divisions are performed recursively as one moves down the hierarchy, connected with GCNs (HDC+GCN); (3) modularity-based community detection where an optimization model is proposed to measure the structure of graphs (Fortunato, 2010; Xiao et al., 2017), connected with GCNs (MOD+GCN); (4) METIS graph partitioning (LaSalle et al., 2015) connected with GCNs (METIS+GCN); (5) feed-forward neural network, connected with GCNs (Xiao et al., 2019) (NN+GCN). In addition, we compare the PGL framework in terms of the application performance with the following baselines: (1) threads in parallel programming (PAR); (2) modularity based community detection to partition the graph into clusters and a heuristic mapping (Xiao et al., 2017) (CommDet); (3) sliding window based neural network to locate specialized structures with a reinforcement learning based mapping (NN+RL) (Xiao et al., 2019); (4) gem5-aladdin, an end-to-end SoC simulation (Shao et al., 2016).

4.1 Graph Learning on Device Mapping Problem

Neural Architecture Search. In this section, we explore different graph neural network architectures (GCN, GAT, and GGNN) on the accuracy of the prediction results. We use 5-fold cross validation by splitting the dataset into five folds, four of which are used as the training data and one used as the testing data for the final accuracy. We vary the number of hidden layers and the number of neurons in a hidden layer to explore the architecture which provides the best accuracy on the testing data. We train each model with 500 epochs with a learning rate of $\alpha = 10^{-3}$, a dropout rate of 0.1, and the batch size of 64. As we can see in Figure 2, the GCN with two hidden layers and 32 neurons per layer gives the 86.33% accuracy whereas GAT with

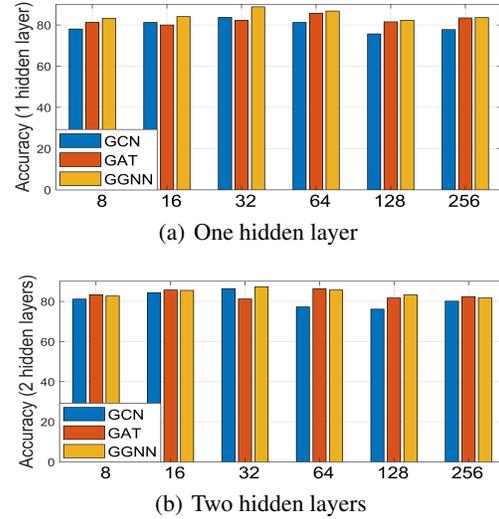


Figure 2. Graph neural network architecture search. We vary the number of neurons in the hidden layer and the number of hidden layers to explore the architecture that gives the best accuracy. GCN, GAT, and GGNN can give 86.33%, 86.5%, and 88.75% accuracy, respectively. Therefore, due to shorter training and testing time, we decide to use GCN as our graph learning model for the next experiments.

Table 4. We compare the graph models used in the PGL framework with state-of-the-art graph representations for programs including DeepTune, NCC, and PROGRAML. Our framework can provide up to 88.75% accuracy compared to PROGRAML.

Framework	Accuracy	Precision	Recall	F_1
DeepTune	68.4%	0.70	0.68	0.69
NCC	78.5%	0.79	0.79	0.79
PROGRAML	80.0%	0.81	0.80	0.80
PGL-GCN	86.33%	0.86	0.85	0.85
PGL-GAT	86.5%	0.87	0.87	0.87
PGL-GGNN	88.75%	0.89	0.89	0.89

two hidden layers and 64 neurons per layer, GGNN with one hidden layer and 32 neurons per layer provide 86.5% and 88.75%, respectively. Therefore, we think that three different graph learning models can give relatively the same accuracy. However, due to the fact that the required training and testing time for GAT and GGNN is substantially longer compared to GCN, we decide to use the GCN with two hidden layers and 32 neurons per layer as our next model to validate the overall PGL framework.

Graph Models. Except for comparing the different graph feature embeddings, we also compare the graph model with some state-of-the-art techniques on graph representations for programs in terms of the accuracy of the prediction results on the same dataset (Cummins et al., 2017). As we can see from Table 4, DeepTune and NCC can only give less than 80% accuracy whereas PROGRAML provides up to 80% accuracy on the Nvidia dataset. However, according to neural architecture search, our graph learning models,

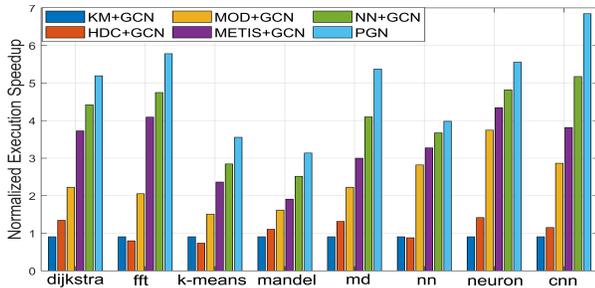


Figure 3. Graph partitioning algorithm. We compare the graph partitioning GAE with different traditional algorithms including K-means, hierarchical divisive clustering, modularity-based community detection, METIS, feed-forward neural network, and GAE, and measure the normalized application execution speedup to validate the benefits of the GAE.

i.e., GCN, GAT, and GGNN, can provide up to 88.75% accuracy compared to others. The reason is that our graph representation of a program differs from the same program with different number of iterations in *for* loops.

4.2 Graph Partitioning

In this section, in order to validate the advantages of the GAE used to partition the large input application into small kernels in the PGL framework, we fix the graph neural network as GCN with two hidden layers and 32 neurons per layer, which is used to predict the correct label for each kernel. We compare the GAE with different partitioning algorithms such as K-means (KM), hierarchical divisive clustering (HDC), modularity-based community detection (MOD), METIS, and feed-forward neural network (NN) in terms of the total application execution speedup. As shown in Figure 3, for the partitioning models without machine learning such as KM, HDC, MOD, and METIS, the normalized execution speedup is smaller compared to the learning models such as NN and GAE. It is mainly because the kernels after graph partitioning are not well recognized by the GCN model. For the learning models, GAE outperforms NN by up to 32% in a sense that the GAE takes into account the graph structures of code.

4.3 The PGL Framework

The proposed PGL framework is able to predict which code segments run best on a specific processor. Therefore, in order to validate the framework including the GAE and GNN models, we use the trained models discussed in Section 4.1 to predict each application in Table 2. As shown in Figure 4, we use the traditional thread based parallel programming running on CPUs as our baseline and compare the PGL framework with community detection, neural network with reinforcement learning, and gem5-aladdin. We observe that the PGL framework can provide up to 6.22x speedup compared to the baseline and 1.91x speedup higher compared to

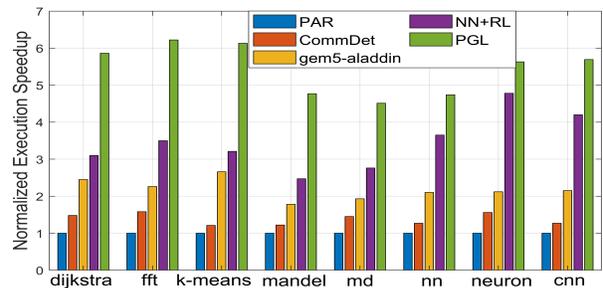


Figure 4. The PGL framework. We compare the framework with the traditional thread based parallel programming, modularity based community detection, neural network with reinforcement learning based mapping, and gem5-aladdin in terms of the application performance. We conclude that our approach can achieve 1.91x better compared to the state-of-the-art techniques.

the state-of-the-art neural network.

5 CONCLUSION

We proposed an end-to-end learnable PGL framework to predict which code segments run best on a specific hardware device. We first transform each application into a dynamic dataflow graph. Next, we build a graph auto encoder (GAE) and spectral clustering to find cluster partition from the distance matrix. We then use GNNs as the learning model to predict the type of each cluster. Our evaluation on 32 CPUs and 32 GPUs concludes that the PGL framework can provide up to 6.22x speedup compared to the baseline and 1.91x higher speedup compared to state-of-the-art techniques.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the support by the National Science Foundation under the Career Award CPS/CNS-1453860, the NSF award under Grant Numbers CCF-1837131, MCB-1936775, and CNS-1932620, and the DARPA Young Faculty Award and DARPA Director Award, under Grant Number N66001-17-1-4044, and a Northrop Grumman grant. The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied by the Defense Advanced Research Projects Agency, the Department of Defense or the National Science Foundation.

A NOTICES AND DISCLAIMERS

Intel technologies may require enabled hardware, software or service activation. No product or component can be absolutely secure. Your costs and results may vary. © Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

REFERENCES

- Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
- Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., and Silvano, C. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.
- Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336*, 2018.
- Brauckmann, A., Goens, A., Ertel, S., and Castrillon, J. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, pp. 201–211, 2020.
- Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232. IEEE, 2017.
- Cummins, C., Fisches, Z. V., Ben-Nun, T., Hoefler, T., and Leather, H. Programl: Graph-based deep learning for program optimization and analysis. *arXiv preprint arXiv:2003.10536*, 2020.
- Fortunato, S. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.
- Grewe, D., Wang, Z., and O’Boyle, M. F. 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.
- Haj-Ali, A., Ahmed, N. K., Willke, T., Gonzalez, J., Asanovic, K., and Stoica, I. A view on deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275*, 2019.
- Haj-Ali, A., Ahmed, N. K., Willke, T., Shao, Y. S., Asanovic, K., and Stoica, I. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 242–255, 2020.
- Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.
- Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.
- Jinnai, Y., Mehrjou, A., Ciosek, K., Mitenkova, A., Lawrence, A., Ellis, T., Tomioka, R., Jones, S. P., and Fitzgibbon, A. Knossos: Compiling ai with ai. 2019.
- Kipf, T. N. and Welling, M. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.
- Krishnan, S., Borojerdian, B., Fu, W., Faust, A., and Reddi, V. J. Air learning: An ai research platform for algorithm-hardware benchmarking of autonomous aerial robots. *arXiv preprint arXiv:1906.00421*, 2019.
- LaSalle, D., Patwary, M. M. A., Satish, N., Sundaram, N., Dubey, P., and Karypis, G. Improving graph partitioning for modern graphs and architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pp. 1–4, 2015.
- Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., and Qian, D. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.
- Li, Y., Gu, C., Dullien, T., Vinyals, O., and Kohli, P. Graph matching networks for learning the similarity of graph structured objects. In *International Conference on Machine Learning*, pp. 3835–3845. PMLR, 2019.
- Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- Nguyen, A. T., Nguyen, T. D., Phan, H. D., and Nguyen, T. N. A deep neural network language model with contexts for source code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 323–334. IEEE, 2018.
- Shao, Y. S., Xi, S. L., Srinivasan, V., Wei, G.-Y., and Brooks, D. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. IEEE, 2016.
- Von Luxburg, U. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.
- Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 2020.

Xiao, Y., Xue, Y., Nazarian, S., and Bogdan, P. A load balancing inspired optimization framework for exascale multicore systems: A complex networks approach. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 217–224. IEEE, 2017.

Xiao, Y., Nazarian, S., and Bogdan, P. Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(6):1416–1427, 2019.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

Zhou, Y., Roy, S., Abdolrashidi, A., Wong, D., Ma, P., Xu, Q., Liu, H., Phothilimtha, M. P., Wang, S., Goldie, A., et al. Transferable graph optimizers for ml compilers. *arXiv preprint arXiv:2010.12438*, 2020.