Conference Paper Review

of

“Compiler-based graph representations for deep learning models of code”

Authors: Alexander Brauckmann| Andrés Goens| Sebastian Ertel | Jeronimo Castrillon

Published in: CC 2020: Proceedings of the 29th International Conference on Compiler Construction

Group Members:

Varun Kumar
2K19/IT/140

Yashit Kumar
2K19/IT/149

Subject: Compiler Design (IT302)

Submitted To: Sunakshi Mehra Ma’am

Submission Date: 24 April 2022

INDEX

ABSTRACT

In natural language processing, novel methods in deep learning, like recurrent neural networks (RNNs) on sequences of words, have been very successful. In contrast to natural languages, programming languages usually have a well-defined structure.

With this structure compilers can reason about programs, using graphs such as abstract syntax trees (ASTs) or control-data flow graphs (CDFGs). In this paper, we argue that we should use these graph structures instead of sequences for learning compiler optimization tasks.

To this end, we use graph neural networks (GNNs) for learning predictive compiler tasks on two representations based on ASTs and CDFGs. Experiments show that this improves upon the state-of-the-art in the task of heterogeneous OpenCL mapping, while providing orders of magnitude faster inference times, crucial for compiler optimizations. When testing on benchmark suites not included for training, our AST-based model significantly outperforms the state-of-the-art by over 12 percentage points in terms of accuracy.

It is the only one to perform clearly better than a random mapping. On the task of predicting thread coarsening factors, we show that all of the methods fail to produce an overall speedup.

INTRODUCTION

The last decade has seen tremendous improvements in machine learning, especially due to deep learning methods, which do not rely on manually-specified features of the data to be learned, but are able to learn what features are important on their own.

Deep learning methods have revolutionized several fields like image recognition or natural language processing. While progress has been made in compiler optimization and tasks related to programming languages, deep learning still plays a comparably modest role in these fields. Most approaches to deep learning in compiler optimization borrow ideas from the successful deep learning methods in natural language processing. This is very reasonable, as the similarities between natural languages and programming languages have fueled cross-fertilization of these two fields for many years.

However, the nature of the analysis required for compiler optimization has shown to require very specific structural properties, which might be neglected by the sequential nature of these methods. More generally, compilers base most analysis on non-sequential data structures like abstract syntax trees (ASTs) and control- and dataflow graphs (CDFGs).

We argue that these kinds of structures are very different from the sequences of words that compose natural language. Data dependencies can be crucial on parts of the code that are far apart in the code string, and details in a small part of the source, like the precise array indices in loops, can have a tremendous impact on some optimizations. For this reason, focusing on the similarities to natural language processing for deep learning in compilers might be ill-advised.

In this paper, we propose to re-evaluate the representations of code we use in deep learning for compilers

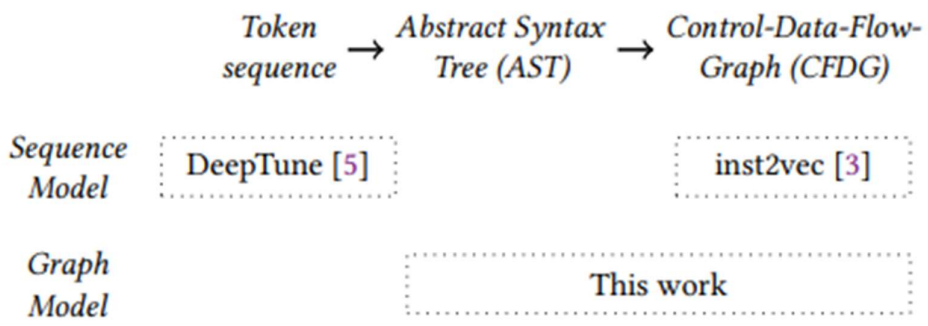

Figure 1. Representations of code in the compiler with their respective machine learning models.

Theoretical Background

A compiler typically uses several representations of code at different stages in order to enable different analyses and optimizations.
we will discuss three of the most common representations at the different stages and how we expect their properties to affect machine learning tasks.

Sequence of Tokens

Conceptually, a compiler first turns a string of characters into a sequence of tokens. Figure 2 shows the example kernel as a sequence of tokens.

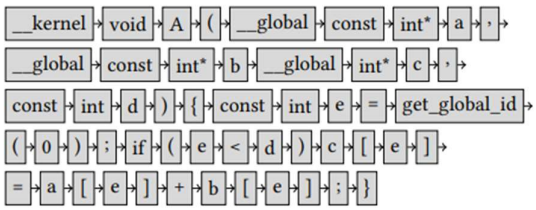

Figure 2. An example kernel as a sequence of tokens.

Abstract Syntax-Tree

After parsing, an abstract syntax tree (AST) on its own loses some information. In particular, if an AST does not include the identifier strings, then it is impossible to tell when two identifiers refer to the same data.

To deal with this, instead of embedding the identifiers, we enhance the AST with (labeled) dataflow edges, which connect two nodes in the AST that refer to the same data.

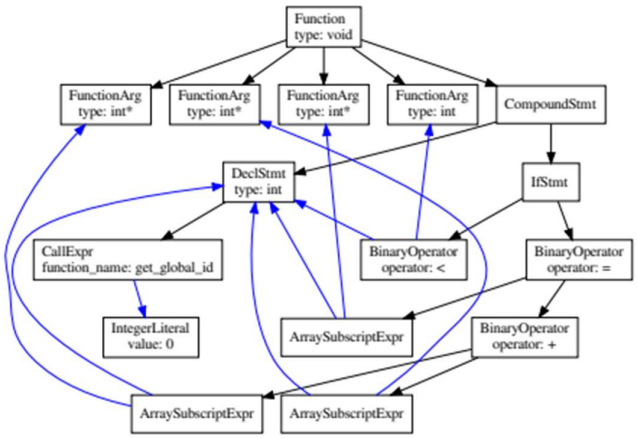

Figure 3. An example kernel as an AST with overlayed dataflow edges (in blue).

Control- and Dataflow Graph

Finally, the control and data flow graph (CDFG). This graph organizes the statements in the code not by their grammar, but by the semantics of the possible flow of control in the program. The basic structure of the CDFG is that of a graph. This graph will usually contain cycles, e.g. if the code contains loops. Similar to the AST+DF, we define the following representation:

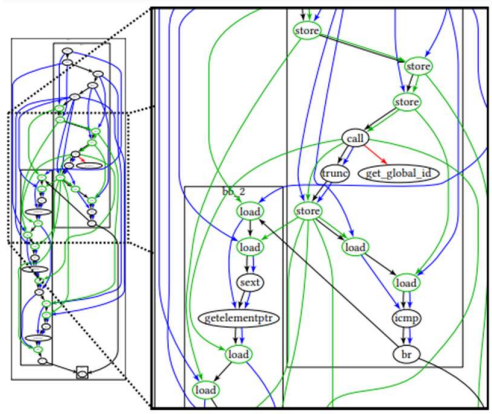

Figure 4. An Example Kernel as a CDFG. The control flow is depicted by black arrows, the data flow by green ones, the red arrow represents an external function call.

Machine Learning Architectures

As this paper focuses on predictive models, we first explain some core principles for ML-driven predictive models in compiler.

Predictive Models

Predictive models revolve around a simple principle: based on some input data X, predict some output data Y. In the context of compiler optimizations, the input data X is the program code, and perhaps additional information about the task

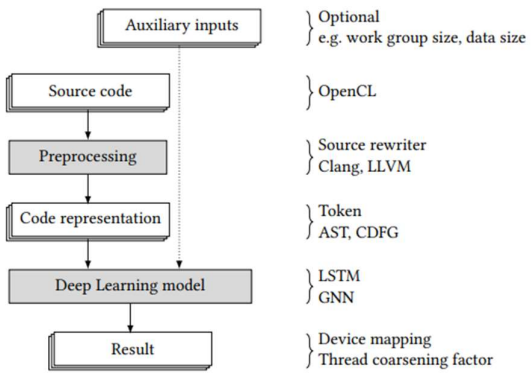

Figure 5. High-level overview of the predictive task.

For the deep learning component in sequential-models, as the token-based model used well-known recurrent neural network architectures like LSTM can be used as representation models. For graph-based models, however, we need to replace the ML architecture used in the flow. In the following, we describe the architecture of a deep learning component of this flow based on graph neural networks. We use this architecture to perform predictive tasks on the AST and CDFG structure

Graph Neural Networks

The input structure to the model is a labeled graph G = (V, E). The output of the model is an n-sized vector representing a probability distribution, with n being the number of classes.

We transform the graph representations of code into the input structure of the model by generating annotated types, considering the graphs of the whole dataset.

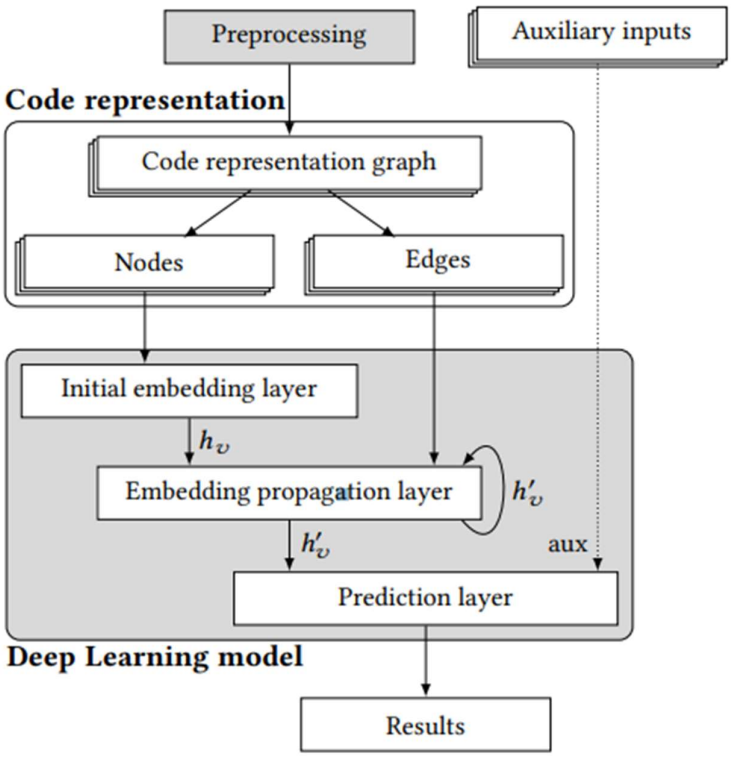

Figure 6. An overview of the predictive model architecture based on graph neural networks.

Evaluation

In this section, we present an evaluation of our methods using the different representations of code. We do this by using two different complex tasks on OpenCL kernels.

In these experiments, we observe a variance in the results for all models, as the initialization of their weights is random and the model converges to different minima. Therefore, we repeat the execution of each experiment 10 times and report aggregated results.

CPU/GPU Mapping

A correctly predicted mapping of a sample yields a faster execution, resulting in a speedup over a static mapping. In a static mapping, a single platform (CPU/GPU) is selected and all kernels are mapped to this platform. To select, the platform which is fastest in most of the training samples for all test benchmarks is chosen.

Additionally, we compare the models in terms of the number of trainable parameters, which describe their sizes, as well as their training and inference time. Inference time is particularly important for compiler-related tasks, as it translates to a faster compilation time for an end-user.

Experimental Setup - The dataset consists of the seven benchmark suites AMD SDK, NPB, NVIDIA SDK, Parboil, Polybench, Rodinia, and SHOC along with the execution times for both CPU and GPU on two different heterogeneous systems, one with an AMD Tahi 7970 GPU and one with an NVIDIA GTX 970 GPU.

We compare both of our methods to the state of the art methods DeepTune and inst2vec . We also compare it to the decision-tree-based model of Grewe et al . Additionally, we compare it to the static mapping model, and to a model choosing CPU or GPU at random.

Table 1 gives a comparison of the network sizes, in terms of the number of trainable parameters, as well as the training and inference times. We can see that Grewe et al's model is considerably faster in training and inference and has the least amount of trainable parameters. The deep learning models are larger in size by several orders of magnitude. **It is important that the AST-based model is an order of magnitude faster in inference than the other deep-learning based models**

Table 1. Model sizes for the device mapping task

Model	Code Representation	Model architecture	No. of learnable parameters	Training time	Inference time
Grewe et al. [17]	Manual features	Decision tree	14	1.02×10^{-3} s	8.46×10^{-5} s
DeepTune [5]	C tokens	Recurrent neural network	76908	6.72×10^{2} s	9.02×10^{-1} s
inst2vec [3]	LLVM-IR tokens	Recurrent neural network	649372	1.08×10^{3} s	1.34 s
GNN-AST	AST+DF	Graph neural network	139846	8.40×10^{2} s	9.95×10^{-2} s
GNN-CDFG	CDFG+CALL+MEM	Graph neural network	89798	1.59×10^{3} s	9.8×10^{-1} s

Figure 8. Accuracy results compared to state-of-the-art.

Figure 10 shows the results of the experiment with this alternative setup. Since we split the train and test sets by benchmarks, we can see how the models fare on every benchmark after being trained on the other six. It is notable how the different methods can have vastly different results on
the different benchmarks. The final entries for each platform show an aggregated result over all benchmarks (artithmetic mean).

We can see that GNN-AST is not only the one with the best overall results, but also the most consistent ones. It had an overall accuracy of 60.6%, which is over 12 percentage points better than that of DeepTune at 47.9% and even better than the 44.9% overall accuracy obtained by inst2vec. In this case, GNN-CDFG had a similar performance, beating DeepTune by less than a percentage point in accuracy at 48.5%. In fact, we see how when tested on different benchmark suites than they were trained on, all state-of-the-art methods we compared to here perform worse than a coin-toss (50.9% accuracy). This indicates that the models probably do not learn the relationship between the code's semantics and the optimal compute device in a way that is generalizable when the code becomes different enough

Thread Coarsening

In parallel architectures, faster executions can be achieved by merging multiple parallel threads in certain situations. The thread coarsening factor is a parameter that controls this behavior in OpenCL.

Table 2. Model sizes for the thread coarsening task.

Model	Code Representation	Model architecture	No. of learnable parameters	Training time	Inference time
Magni et al. [15]	Manual features	MLP		8.63 s	2.62×10^{-4} s
DeepTune [5]	C tokens	Recurrent neural network	76838	8.66×10^{-1} s	5.59×10^{-1} s
inst2vec [3]	LLVM-IR tokens	Recurrent neural network	649030	1.92×10^{1} s	2.08×10^{-1} s
GNN-AST	AST+DF	Graph neural network	914	4.83 s	1.21×10^{-3} s
GNN-CDFG	CDFG+CALL+MEM	Graph neural network	1024	5.21 s	1.32×10^{-3} s

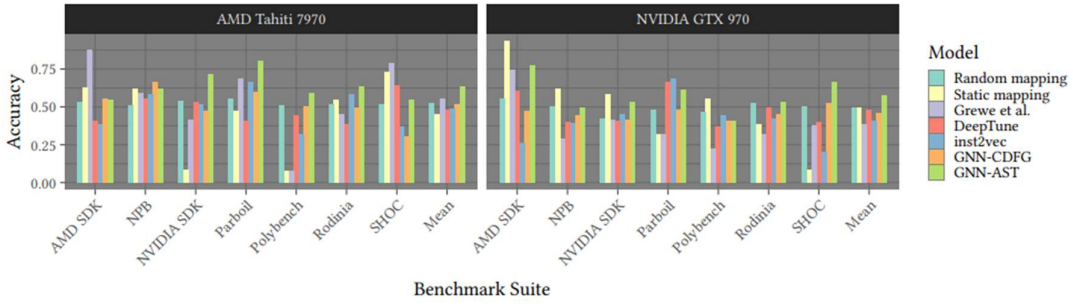

Figure 10. Accuracy of device mapping in grouped setting.

Again, we compare the model sizes and training and inference times of the different models. The results can be seen in Table 2. And again here we see how inference in GNN-based models is orders of magnitude faster than the LSTM-based sequential counterparts.

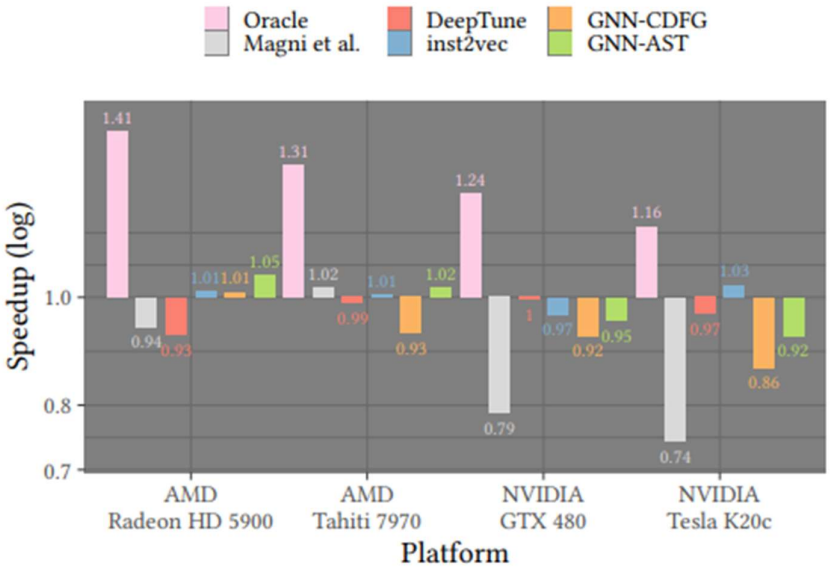

Figure 11. Speedup results in thread coarsening task.

Figure 11 shows the results of the thread coarsening experiment. In the figure we add an łOraclež for reference, which depicts the best possible speedups. As we can see in the figure, the sequence-based models perform better than both graph-based

models in three of the platforms. It is notable, however, that in most cases the predicted thread-coarsening factors yield an overall slowdown. Overall, DeepTune yields an overall speedup (geometric mean) of around 0.97 across all platforms, GNN-AST is slightly better with 0.98 and GNNCDFG is slightly worse with 0.93. The Magni et al. model yields an overall speedup of 0.87. The best results in this task are achieved by inst2vec, which has an overall speedup of 1.00, i.e. just as good as doing nothing. In general, all deep learning methods fare comparably bad at this task. This might be explained in part by the modest possible maximal speedups, which cap at 1.28 overall. However, it is perhaps also an indication that current deep learning models of code are not yet up to this task

Comparison of Graph-Based Models

The graph-based representations we have used for deep learning in this paper feature several enhancements, like the dataflow edges enriching the AST. Similarly, for the CDFG we included edges connecting the corresponding instructions for function calls and the data in load and store instructions. In this section, we want to use the three experimental setups described above to compare how our model fares with and without these enhancements. For this, we used two versions of our AST-based representation, with (AST+DF) and without the dataflow edges (AST).

We trained the model using similar configurations as in the previous experiments. Figure 12 shows the results with the different versions of our graph-based representations. In the random setup for device mapping, all variants of the CDFG based representation fared similarly well, yielding the best results overall. In the grouped setup, we see how the AST based representation yields better results. All experiments, the grouped split mapping especially, show that the addition of the dataflow edges to the AST was fruitful. Finally, as discussed above, the results on the thread coarsening task are pretty modest across all deep learning models.

From these experiments, we cannot conclude that there is a single best compiler-based representation of code for deep learning models. In the random split of the mapping task, where the training data resembles more the test data, the models closer to the execution semantics (CDFG) had the best performance. On the other hand, with the grouped split, where a higher degree of generalization is required, the AST with its more abstract semantics, closer to the programmer, performed better. For the thread coarsening task, which has very little data points available, the non-end-to-end approach of inst2vec fared best, albeit also not particularly well, yielding no speedup overall

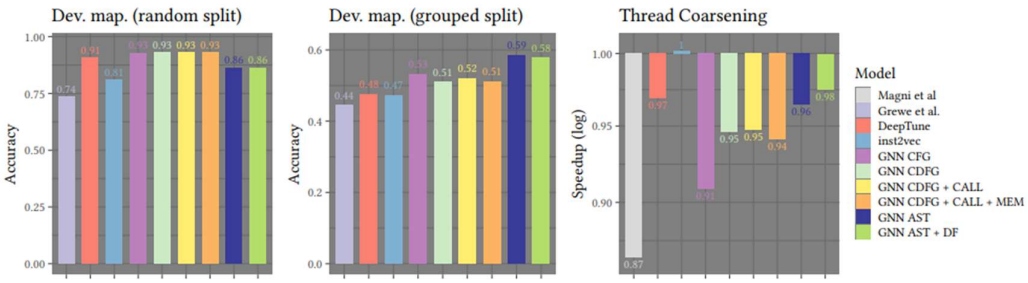

Figure 12. Comparison of graph-based models with different enhancements.

Conclusion

In this paper, we have compared code representations for deep learning based on the representations used in the compiler: sequences of tokens, abstract syntax trees, and control and data flow graphs. We have shown different graph based methods to be better suited for a complex classification task, identifying the optimal CPU/GPU mapping for OpenCL kernels. In particular, we outperformed state-of-the-art approaches using sequential models based on token sequences. We also showed more generalization capabilities by splitting training and test data into sets coming from disjoint benchmark suites instead of random subsets of all kernels. In this alternative setup, our graph-based model GNN+AST performed significantly better than their sequential counterparts, which failed to beat a coin-toss

From our experiments, we cannot conclude that there is a single best compiler-based representation of code for deep learning models. In the random split of the mapping task, where the training data resembles more the test data, the models closer to the execution semantics (CDFG) had the best performance. On the other hand, with the grouped split, where a higher degree of generalization is required, the AST with its more abstract semantics, closer to the programmer, performed better. For the thread coarsening task, which has very little data points available, the non-end-to-end approach of inst2vec fared best, albeit also not particularly well, yielding no speedup overall.

The CPU/GPU classification task is a complex task where the interaction between different parts of the code and the target architecture all play a role. Thus, this task serves as a good first challenge for compiler analysis methods. However, the related predictive task of finding optimal thread coarsening factors for OpenCL kernels proved to be too challenging for both sequential and graph-based models. The results of this task indicate that while useful, deep learning methods still struggle on complex compiler-related tasks.

Future Work : We believe our results encourage further investigation of compiler-based representations of code.
Programming languages and machine semantics being strict and structured as they are should also be an advantage in terms of learning, and we can take advantage of decades of research into compiler methods to improve machine learning methods in this domain.
Furthermore, OpenCl kernels are comparatively small. In future work, we plan to investigate these methods on larger code fragments, where we believe the graph structures should be even better suited to track long-range dependencies between parts of the code.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[3] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. Neural code comprehension: a learnable representation of code semantics. In *Advances in Neural Information Processing Systems*, pages 3585–3597, 2018.

[4] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[5] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2017)*, September 2017.

[6] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[7] A. Goens, A. Brauckmann, S. Ertel, C. Cummins, H. Leather, and J. Castrillon. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, MAPL 2019, pages 38–46, New York, NY, USA, June 2019. ACM.

[8] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] D. D. Johnson. Learning graphical state transitions. In *ICLR*, 2017.

[10] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[11] H. Leather, E. Bonilla, and M. O'boyle. Automatic feature generation for machine learning–based optimising compilation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):14, 2014.

[12] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

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

[14] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[15] A. Magni, C. Dubach, and M. O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 455–466. ACM, 2014.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[17] M. F. O'Boyle, Z. Wang, and D. Grewe. 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)*, pages 1–10. IEEE Computer Society, 2013.

[18] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[19] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[20] N. Vasilache, C. Bastoul, and A. Cohen. Polyhedral code generation in the real world. In A. Mycroft and A. Zeller, editors, *Compiler Construction*, pages 185–201, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[21] Z. Wang and M. OâĂŹBoyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[22] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

Link to source paper - https://dl.acm.org/doi/10.1145/3377555.3377894