Towards intelligent compiler optimization

Mihael Kovac*, Mario Brcic*, Agneza Krajna*, Dalibor Krleza*
University of Zagreb Faculty of Electrical Engineering and Computing, Zagreb, Croatia*
mihael.kovac@fer.hr, mario.brcic@fer.hr, agneza.krajna@fer.hr, dalibor.krleza@fer.hr

***Abstract*—The future of computation is massively parallel and heterogeneous with specialized accelerator devices and instruction sets in both edge- and cluster-computing. However, software development is bound to become the bottleneck. To extract the potential of hardware wonders, the software would have to solve the following problems: heterogeneous device mapping, capability discovery, parallelization, adaptation to new ISAs, and many others. This systematic complexity will be impossible to manually tame for human developers. These problems need to be offloaded to intelligent compilers. In this paper, we present the current research that utilizes deep learning, polyhedral optimization, reinforcement learning, etc. We envision the future of compilers as consisting of empirical testing, automatic statistics collection, continual learning, device capability discovery, multiphase compiling – precompiling and JIT tuning, and classification of workloads. We devise a simple classification experiment to demonstrate the power of simple graph neural networks (GNNs) paired with program graphs. The test performance demonstrates the effectiveness and representational appropriateness of GNNs for compiler optimizations in heterogeneous systems. The benefits of intelligent compilers are time savings for the economy, energy savings for the environment, and greater democratization of software development.**

***Keywords*—compiler optimization, GNN, reinforcement learning, edge computing, heterogeneous computing, polyhedral model, machine learning**

I. Introduction

The future of computation is massively parallel and heterogeneous with specialized accelerator devices and instruction sets in both edge- and cluster-computing. With a multitude of different domain-specific accelerators [1]–[3], memory technologies [4]–[6], and different types of processor architectures [7], the engineering effort required to extract the potential of upcoming hardware wonders is becoming an intractable problem [8].

To efficiently use the major potential of the many new types of hardware, prevent software development from becoming a bottleneck, but also improve the efficiency of hardware research itself, we believe that many of the new arising problems should be off-loaded to next-generation, intelligent compilers. These compilers would have to solve the following problems: heterogeneous device mapping, device capability discovery, code partitioning, automatic parallelization, adaptation to new hardware, and many others. The growing systematic complexity will be impossible to manually tame for human developers. These types of compilers would not only further improve software development but also propel hardware research forward. The deployment of actual code which needs to be tested and compared on different kinds of hardware would be shortened, eliminating the need for hardware researchers to write optimal code for the specific devices they tailor.

The earliest compiler optimizations which are still in use today, such as dead code elimination, peephole optimization, function inlining, and similar, mostly relied on some form of a graph representation of a program and a hand-tuned heuristic searching through it to find the appropriate optimization [9]. What we will call the second generation of compiler optimizations, which are still very prevalent and are being researched today, came with the polyhedral model. The polyhedral model abstracts sequential parts of a program into polyhedra and tries to solve an integer linear program (ILP) defined by it [10]–[12], thus finding an optimal schedule of the instructions which are considered in the polyhedra without changing the semantics of the code. The next or third generation of compilers are based on the methods of previous generations but augmented with machine learning methods [13]–[21].

In Section II, we discuss the related works on compiler optimizations, with a focus on the up-and-coming machine learning approaches and the polyhedral model. In Section III, we further describe the problems next-generation compilers have to solve and propose a set of high-level solutions to those problems. Next, in Section IV, we experimentally demonstrate the effectiveness of simple graph neural networks (GNNs) in encoding computational patterns and finally conclude our work in Section V.

II. Related work

A. Polyehdral model

The polyhedral model is a model for optimizing loops in programs which we believe is the first step towards modern, intelligent compilers and automatic code deployment. The model tries to find the optimal schedule of a sequential part of a program by abstracting it into polyhedron. The optimization task then reduces to finding an optimal schedule without changing the semantics of the program. It has to solve an Integer Linear Program (ILP), which is a combinatorial optimization problem, where we try to find the optimal transformations to be applied on a schedule tensor. The model is extensively used in many general-purpose and domain-specific language compilers, such as GCC [22], [23], LLVM [24], TVM [25], Tiramisu [26], MLIR [27] and has seen many new improvements and use cases over the years, such as data-stream processing [28], non-affine code optimization [29] compilation for heterogeneous targets [30], automatic parallelization [31] and many more [32]–[38]. We believe that the polyhedral

model has its place in compiler optimizations, but its future lies in machine learning augmentations. Recent advances in the field have been going in this direction. Polyhedral descriptions can be learned [39] and solving ILPs can be sped up [40], [41]. The relatively recent PolyGym [14] is a reinforcement learning framework for polyhedral optimizations of programs.

However, some limitations of the model also have to be addressed. The first limitation of the model is that it can not represent explicit heterogeneous device mapping, at least in its current form. It can be extremely useful in optimizing code for a specific device, but it can not explicitly map a part of code to it. Secondly, the current optimization targets, i.e. the actual objective functions are predefined for each device. While some work is being done in the generalization of these objective functions [42], the actual heuristics used are still hand-tuned. We address this problem in Section III-A.

B. Manual-heuristic approaches

Some of the earliest compiler optimizations methods rely on aggressively gathering data about a program's behavior to be able to find the best way to optimize it. Most of these methods try to use different graph representations of a program. These approaches, such as finding dominator and post-dominator trees, constant folding, dead-code elimination, reachability, and liveness analysis [9], [16], rely on finding different characteristics of a program graph representation and accordingly optimizing, guided by a hand-tuned (manual) heuristic.

C. Machine learning approaches

A plethora of machine learning approaches in compiler optimization and source code analysis has recently started to emerge. Translating between programming languages [13], detecting code clones [20], [43]–[45], heterogeneous device mapping [13], [16], [46] and other problems. Most of these would be out of scope for manual-heuristic approaches or the polyhedral model, but with different code representations [13], [16], [17], [20], [45]–[47] and intelligent, robust models which can make use of them, a lot of new directions for compilers and program analysis tools have emerged. Specifically, recent graph-based approaches have tried to leverage GNN models to embed the program representation into a continuous space where certain, known optimizations, such as gradient descent can be used to find new optimizations or augment older compilation methods [16], [19]–[21], [44], [45].

Other, syntax-based approaches, have tried to use natural language processing (NLP) methods, where they try to leverage a program's syntax to solve some optimization tasks. While some NLP principles have been adopted, such as using embeddings [13], [46], [47], most of them do not generalize very well when trained on source code. The drawback of NLP models is the focus on syntax, which makes the models much more complex. An NLP model first has to learn how to parse the language into a suitable

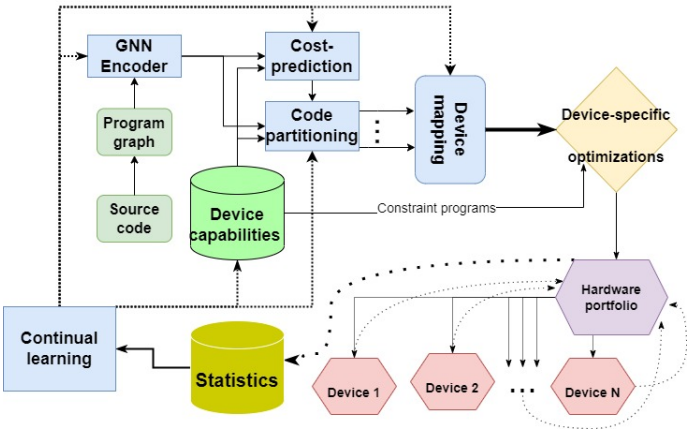

Fig. 1: Intelligent compiler system

representation before doing the actual optimizations on it. This representation needs to be able to encode the sequences of computations, but also the long-range dependencies which are often present in programs. The natural way of representing such sequences and dependencies is with graphs, which we can already extract from programs manually. Therefore, we can avoid the first part of the NLP problem and focus on the optimizations by directly working with graph representations. However, some utility can be found in NLP models, one of which we describe in Section III-B.

Modern machine learning models have reduced comprehensibility and they must be coupled with explainability methods to enable faster innovation and debugging [48]–[50]. Of special interest for compilers are new explainable approaches for supervised and reinforcement learning over GNNs. Additionally, AI methods are subject to limits [51] which may affect the security and performance, both aspects that need further inquiry.

III. Roadmap towards intelligent compilers

Currently, most modern compilers focus on a specific set of optimizations, which can be domain-specific [25], [26], [52], [53] or even device-specific [3], [54]. This limits their viewpoint substantially, as we believe the focus should be on general compiler optimizations and eliminating hand-tuned heuristics from different parts of compilation systems. Most applications today are larger systems, consisting of multiple domain-specific tasks which need to be optimized. Our goal is to provide a vision of generalized compilers of the future. Instead of domain-specific languages and optimizations, we believe that the compiler itself should be able to identify which optimizations are best performed on specific parts of code, as the domain-specific focus can limit cross-domain optimization opportunities. In this section, we present our view of next-generation intelligent compilers by addressing some of the problems we mentioned in the previous sections. Our approach (see 1) focuses on using graph neural networks (GNN) and reinforcement learning to find the best optimization on a multitude of specified problems. We present our view

through a hierarchy of intelligent models and methods which need to work in tandem.

A. Device capability discovery and device-specific optimizations

The first problem we will address is the device capability discovery problem. This is the problem of finding out the set of instructions and/or operations a device is capable of executing. Here we rely on the Bring Your Own Codegen [55] principle. With this approach, we rely on a generalized IR language [56] or even a framework such as OpenCL [57], and a hardware vendor to write the actual code generation. The compiler would be responsible for translating the general-purpose code into the required language while performing device-specific optimizations in the process. Source-to-source compilers, such as Facebook's TransCoder [58] and R-Stream [59], can be leveraged for code generation, but we turn to the polyhedral model for the actual optimization. We can leverage machine learning methods such as (semi-)supervised and reinforcement learning to find the (constraint) linear programming problem which would best represent the device we are mapping to. Our model would either use gathered statistics from previous code executions on the device, reinforcement learning, or a combination of the two, to search the hyperheuristic space. This would mean we are essentially representing the device's capabilities to leverage the potential locality, parallelism, or any other characteristic of our code for a significant performance gain with a learned objective function and constrained problem.

B. Code partitioning

The problem which naturally arises with the previously mentioned approach is deciding which parts of code we want to translate to a deployable representation. This code partitioning problem could be viewed from multiple viewpoints, however, the most straightforward approach would be to implement an extended variant of the min-cut algorithm on a program graph, where we partition the graph based on a runtime cost prediction. However, with this model, we also need to keep in mind that we will be optimizing the newly deployed code with our previously mentioned polyhedral model. Therefore our model needs to take the learned device capability representation into account, along with the actual program graph.

Another viewpoint on this problem is through the lens of preprocessor directives. Many frameworks, such as OpenMP [60], use preprocessing directives to automatically translate or parallelize parts of code. This problem can be viewed as a type of automatic and intelligent generation of such directives based on learned heuristics over program representations, which would be a syntax-based approach since we need to generate the directives. Alternatively, we can also approach this problem with a Just-In-Time (JIT) compilation model in which case we can use a graph-based approach, therefore bypassing the source code by dynamically injecting functionality augmented with runtime information.

C. Heterogeneous device mapping

Automatic mapping to different heterogeneous devices has been a long-standing problem in the compiler research community. Recently, graph neural networks [16], along with IR instruction embedding techniques [13], [46] were used in automatically mapping OpenCL kernels to either the CPU or GPU. Although these methods show some promising results, they are far from mapping in entire heterogeneous systems, where we can map tasks to a combination of devices as well. Other approaches have tried to leverage the polyhedral model [30] or hand-crafted algorithms [61], while some limit themselves to deep learning workloads [8]. So far the under-utilized approach of graph neural networks may hold the potential, but with certain augmentations to boost its precision and effectiveness. Another augmentation to the model can be added by using hardware portfolios. Analogous to the works on algorithm portfolios [62], the model could deploy the code on a multitude of devices simultaneously and use the result it gets the quickest. This would allow the model to be more fault-tolerant and use the statistical variance of the runtime distribution for better performance when running on different problem instances.

D. Statistics collection, empirical testing and continual learning

To ensure longevity, robustness, and generalization for upcoming hardware architectures, the next-generation compilers need to be highly adaptable. For the compilers to be usable in a similar to GCC or Clang compilers which have stood the test of time, we would have to employ some sort of continual learning methods [63], [64] for our model to be usable through its lifetime. Contrary to classic machine learning methods which learn in isolation on a given dataset, continual learning methods try to steer the model towards continual knowledge accumulation, which it can use in future learning and problem-solving tasks [63]. This means that the knowledge the model has gathered through multiple training runs, possibly driven by combinatorial designs [65], is not only remembered but also reused and transferred to new problems. By collecting statistics of previous compilation and program executions on different hardware platforms we can compare our model to its previous versions. The collected statistics can also be used as a form of regularization of the model so it does not forget the previously learned optimization techniques. Since we are working with statistical models, the model's ability to adapt to new problems or hardware platforms, while also maintaining performance on the old, learned ones, would require a large of empirical tests to ensure the robustness of the model.

E. Precompiling and JIT tuning

A problem with our approach is that every application has to be compiled from a source for it to be optimized for

a platform. This could be circumvented by precompiling our code and then JIT tunning on the platform itself. This would require us to design an annotated bytecode or intermediate representation into which optimal code can be injected. The code could also be precompiled for a family of devices since certain types of computations are always more suited for specific devices. For example, interacting with files would always be done by the CPU, which means there is no need for it to recompile each time we want to deploy to a certain device.

IV. Experimental study

We have put a lot of focus on graph-neural networks in our proposed approach. In this section, we will demonstrate with a simple experiment that GNNs have sufficient representational power and inbuilt bias to solve problems over code graphs. We will focus on the device mapping problem by solving an intermediate problem – extracting characteristics and higher-level patterns from the program graph. Reasoning over those patterns is just a matter of scaling, and it is not covered here. Here we will classify computational dwarfs from the OpenCL kernel using graph neural networks. Dwarfs are reoccurring communication and computation patterns common to a class of applications [66], [67]. We reuse the OpenCL kernels used in the DEVMAP task [13], [16] which are collected from a multitude of open-source benchmarks for GPUs, parallel computing, and heterogeneous systems. After collecting the set of 282 kernels, some are labeled using soft labels as they can belong to multiple dwarfs, while some are omitted since we were not sure of even a single dwarf the kernel would belong to. Very few of the kernels have hard labels (one-hot). After labeling, we split the kernels into train and test data with 80:20% ratio and generate a total of 3784 unique LLVM IR programs, by using the Clang compiler with various combinations of compile flags. With this approach, we aim to increase the size of the dataset, but also to enrich it with noisy data so our trained models become more robust and impervious to it. The distribution of the dataset with regards to the highest probable labels is shown in Figure 2.

We can observe that the dataset is not well-balanced and some classes, which have less than 100 instances have been omitted during training and testing. This stems from the fact that most benchmarks for parallel computing and heterogeneous platforms focus on GPUs and data-level parallelism which is why classes with higher potential for it are more prevalent. Also, some kernels may have been mislabeled as there is room for human error in the labeling process.

We use a simple graph convolutional network model described in [68], with $T = 4$ message passes and $H = 128 + 16$ node representation dimensions. The first $H1 = 16$ dimensions are used to encode the type of the node, which can be an instruction, variable or constant, while the $H2 = 128$ encode the text of the node (IR). We use a vocabulary used in CompilerGym's [15] GNN cost

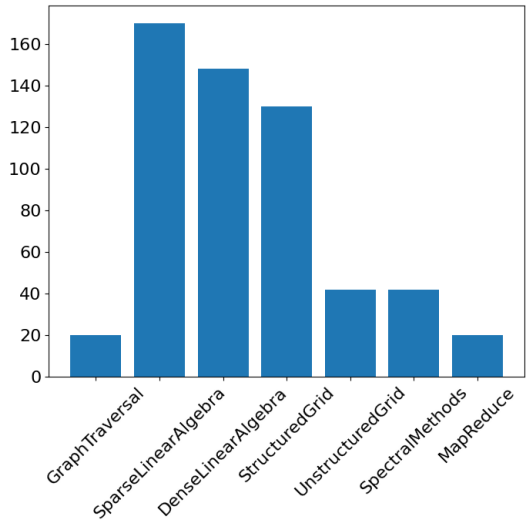

Fig. 2: Dataset distribution.

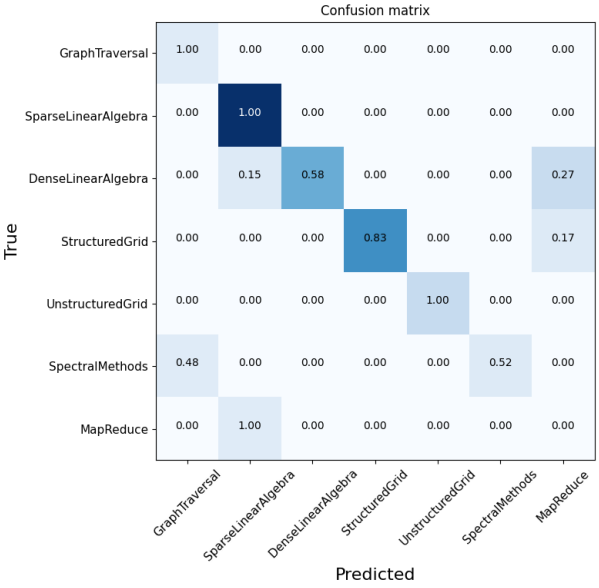

Fig. 3: Test set confusion matrix.

model example, which contains embedding indices and we train our embeddings. This gives us 78.3% accuracy on the test set and the confusion matrix shown in Figure 3. This level of performance demonstrates the effectiveness and representational appropriateness of GNNs for compiler optimizations in heterogeneous systems.

V. Conclusion

Through the experiment, we show that encoding code patterns can be effectively solved using GNNs. However, if we want the next-generation compilers to be as widely used as the general-purpose compilers we currently have, further work needs to be done on specialized GNN architectures for specific purposes and the problems we have proposed, which should efficiently boost the precision of next-generation compilers. Furthermore, automatically deploying optimal programs on every platform

could prove beneficial for the environment by lowering energy consumption [69], [70], the economy by lowering development time, and also software development itself, since the need for manually optimizing code would be mostly eliminated. Just imagine the effects of intelligent compilers on app stores, both from the developers' and customers' views. Similar goes for distributed computing in high-performance clusters where the first steps have already been made for simpler scenarios [8].

REFERENCES

[1] R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss, "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators," *Concurrency and Computation: Practice & Experience*, vol. 24, no. 7, pp. 663–675, May 2012. [Online]. Available: https://doi.org/10.1002/cpe.1904

[2] F. Schuiki, M. Schaffner, and L. Benini, "NTX: An Energy-efficient Streaming Accelerator for Floating-point Generalized Reduction Workloads in 22nm FD-SOI," *arXiv:1812.00182 [cs]*, Dec. 2018, arXiv: 1812.00182. [Online]. Available: http://arxiv.org/abs/1812.00182

[3] G. Zhou, J. Zhou, and H. Lin, "Research on NVIDIA Deep Learning Accelerator," in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, Nov. 2018, pp. 192–195, iSSN: 2163-5056.

[4] S. Jalaleddine, "Associative Memories and Processors: The Exact Match Paradigm," *Journal of King Saud University - Computer and Information Sciences*, vol. 11, pp. 45–67, Dec. 1999.

[5] L. Li, L. Gao, and J. Xue, "Memory coloring: a compiler approach for scratchpad memory management," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, Sep. 2005, pp. 329–338, iSSN: 1089-795X.

[6] R. R. Silva, C. M. Hirata, and J. de Castro Lima, "Big high-dimension data cube designs for hybrid memory systems," *Knowledge and Information Systems*, vol. 62, no. 12, pp. 4717–4746, Dec. 2020. [Online]. Available: https://doi.org/10.1007/s10115-020-01505-9

[7] M. Smotherman, "Understanding EPIC Architectures and Implementations," p. 8.

[8] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning," *arXiv:2201.12023 [cs]*, Jan. 2022, arXiv: 2201.12023. [Online]. Available: http://arxiv.org/abs/2201.12023

[9] "Compilers: principles, techniques, and tools | Guide books." [Online]. Available: https://dl.acm.org/doi/10.5555/6448

[10] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '78. New York, NY, USA: Association for Computing Machinery, Jan. 1978, pp. 84–96. [Online]. Available: https://doi.org/10.1145/512760.512770

[11] M. Griebl, C. Lengauer, and S. Wetzel, "Code Generation in the Polytope Model," in *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[12] W. Pugh, "Uniform techniques for loop optimization," in *Proceedings of the 5th international conference on Supercomputing*, ser. ICS '91. New York, NY, USA: Association for Computing Machinery, Jun. 1991, pp. 341–352. [Online]. Available: https://doi.org/10.1145/109025.109108

[13] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural Code Comprehension: A Learnable Representation of Code Semantics," *arXiv:1806.07336 [cs, stat]*, Nov. 2018, arXiv: 1806.07336. [Online]. Available: http://arxiv.org/abs/1806.07336

[14] A. Brauckmann, A. Goens, and J. Castrillon, "PolyGym: Polyhedral Optimizations as an Environment for Reinforcement Learning," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2021, pp. 17–29.

[15] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," *arXiv:2109.08267 [cs]*, Dec. 2021, arXiv: 2109.08267. [Online]. Available: http://arxiv.org/abs/2109.08267

[16] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, and H. Leather, "ProGraML: Graph-based Deep Learning for Program Optimization and Analysis," *arXiv:2003.10536 [cs, stat]*, Mar. 2020, arXiv: 2003.10536. [Online]. Available: http://arxiv.org/abs/2003.10536

[17] G. Ma, Y. Xiao, M. Capotă, T. L. Willke, S. Nazarian, P. Bogdan, and N. K. Ahmed, "Learning Code Representations Using Multifractal-based Graph Networks," in *2021 IEEE International Conference on Big Data (Big Data)*, Dec. 2021, pp. 1858–1866.

[18] R. Mammadli, M. Selakovic, F. Wolf, and M. Pradel, "Learning to Make Compiler Optimizations More Effective," *arXiv:2102.13514 [cs]*, Feb. 2021, arXiv: 2102.13514. [Online]. Available: http://arxiv.org/abs/2102.13514

[19] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional Neural Networks over Tree Structures for Programming Language Processing," *arXiv:1409.5718 [cs]*, Dec. 2015, arXiv: 1409.5718. [Online]. Available: http://arxiv.org/abs/1409.5718

[20] D. Xiao, D. Hang, L. Ai, S. Li, and H. Liang, "Path context augmented statement and network for learning programs," *Empirical Software Engineering*, vol. 27, no. 2, p. 37, Jan. 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10098-y

[21] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. M. Phothilimthana, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon, "Transferable Graph Optimizers for ML Compilers," *arXiv:2010.12438 [cs]*, Feb. 2021, arXiv: 2010.12438. [Online]. Available: http://arxiv.org/abs/2010.12438

[22] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "GRAPHITE: Polyhedral analyses and optimizations for GCC," Jun. 2006.

[23] K. Trifunovic, A. Cohen, D. Edelsohn, L. Feng, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta, "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation," Jan. 2010.

[24] T. Grosser, H. Zheng, and R. Aloor, "Polly - Polyhedral optimization in LLVM," p. 6.

[25] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," *arXiv:1802.04799 [cs]*, Oct. 2018, arXiv: 1802.04799. [Online]. Available: http://arxiv.org/abs/1802.04799

[26] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code," *arXiv:1804.10694 [cs]*, Dec. 2018, arXiv: 1804.10694. [Online]. Available: http://arxiv.org/abs/1804.10694

[27] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law," *arXiv:2002.11054 [cs]*, Feb. 2020, arXiv: 2002.11054. [Online]. Available: http://arxiv.org/abs/2002.11054

[28] J. Leben and G. Tzanetakis, "Polyhedral Compilation for Multi-dimensional Stream Processing," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 3, pp. 27:1–27:26, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3330999

[29] P. Quinton and T. Yuki, "Representing Non-Affine Parallel Algorithms by means of Recursive Polyhedral Equations," in *IMPACT 2021 - International Microsystems, Packaging, Assembly and Circuits Technology conference*. Budapest, Hungary: Hipeac 2021, Jan. 2021, pp. 1–11. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03518569

[30] T. Grosser and T. Hoefler, "Polly-ACC Transparent compilation to heterogeneous hardware," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 1–13. [Online]. Available: https://doi.org/10.1145/2925426.2926286

[31] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1379022.1375595

[32] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The Polyhedral Model Is More Widely Applicable Than You Think," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Gupta, Ed. Berlin, Heidelberg: Springer, 2010, pp. 283–303.

[33] P. Feautrier, "Automatic Parallelization in the Polytope Model," vol. 1132, Aug. 1996.

[34] T. Grosser, "A decoupled approach to high-level loop optimization: tile shapes, polyhedral building blocks and low-level compilers," p. 225.

[35] A. S. Rajam, "Beyond the realm of the polyhedral model: combining speculative program parallelization with polyhedral compilation," p. 185.

[36] A. Rasch, "md_poly: A Performance-Portable Polyhedral Compiler Based on Multi-Dimensional Homomorphisms," p. 5.

[37] S. Tavarageri, A. Heinecke, S. Avancha, G. Goyal, R. Upadrasta, and B. Kaul, "PolyDL: Polyhedral Optimizations for Creation of High Performance DL primitives," *arXiv:2006.02230 [cs]*, Nov. 2020, arXiv: 2006.02230. [Online]. Available: http://arxiv.org/abs/2006.02230

[38] J. Zhao, "A combined language and polyhedral approach to heterogeneous parallelism," p. 140.

[39] D. Maragno, H. Wiberg, D. Bertsimas, S. I. Birbil, D. d. Hertog, and A. Fajemisin, "Mixed-Integer Optimization with Constraint Learning," *arXiv:2111.04469 [cs, math, stat]*, Nov. 2021, arXiv: 2111.04469. [Online]. Available: http://arxiv.org/abs/2111.04469

[40] M. Gasse, D. Chételat, N. Ferroni, L. Charlin, and A. Lodi, "Exact Combinatorial Optimization with Graph Convolutional Neural Networks," *arXiv:1906.01629 [cs, math, stat]*, Oct. 2019, arXiv: 1906.01629. [Online]. Available: http://arxiv.org/abs/1906.01629

[41] J. Juros, M. Brcic, M. Koncic, and M. Kovac, "Exact solving scheduling problems accelerated by graph neural networks," in *(under review)*, Feb. 2022. [Online]. Available: http://dx.doi.org/10.13140/RG.2.2.19709.23528/1

[42] L. Chelini, T. Gysi, T. Grosser, M. Kong, and H. Corporaal, "Automatic Generation of Multi-Objective Polyhedral Compiler Transformations," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Virtual Event GA USA: ACM, Sep. 2020, pp. 83–96. [Online]. Available: https://dl.acm.org/doi/10.1145/3410463.3414635

[43] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 516–527. [Online]. Available: https://doi.org/10.1145/3395363.3397362

[44] N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, and R. Purandare, "Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks," *arXiv:2011.11228 [cs]*, Nov. 2020, arXiv: 2011.11228. [Online]. Available: http://arxiv.org/abs/2011.11228

[45] G. Zhao and J. Huang, "DeepSim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 141–151. [Online]. Available: https://doi.org/10.1145/3236024.3236068

[46] S. VenkataKeerthy, R. Aggarwal, S. Jain, M. S. Desarkar, R. Upadrasta, and Y. N. Srikant, "IR2Vec: LLVM IR based Scalable Program Embeddings," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 1–27, Dec. 2020, arXiv: 1909.06228. [Online]. Available: http://arxiv.org/abs/1909.06228

[47] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," *arXiv:1803.09473 [cs, stat]*, Oct. 2018, arXiv: 1803.09473. [Online]. Available: http://arxiv.org/abs/1803.09473

[48] F. K. Dosilovic, M. Brcic, and N. Hlupic, "Explainable artificial intelligence: A survey," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija: IEEE, May 2018, pp. 0210–0215. [Online]. Available: https://ieeexplore.ieee.org/document/8400040/

[49] M. Juric, A. Sandic, and M. Brcic, "AI safety: state of the field through quantitative lens," Feb. 2020. [Online]. Available: https://arxiv.org/abs/2002.05671v1

[50] A. Krajna, M. Brcic, M. Kovac, and A. Sarcevic, "Explainable Artificial Intelligence: An Updated Perspective," 2022.

[51] M. Brcic and R. V. Yampolskiy, "Impossibility Results in AI: A Survey," *arXiv:2109.00484 [cs]*, Sep. 2021, arXiv: 2109.00484. [Online]. Available: https://doi.org/10.48550/arXiv.2109.00484

[52] Google, "XLA: Optimizing Compiler for Machine Learning." [Online]. Available: https://www.tensorflow.org/xla

[53] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph Lowering Compiler Techniques for Neural Networks," *arXiv:1805.00907 [cs]*, Apr. 2019, arXiv: 1805.00907. [Online]. Available: http://arxiv.org/abs/1805.00907

[54] A. Yazdanbakhsh, K. Seshadri, B. Akin, J. Laudon, and R. Narayanaswami, "An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks," *arXiv:2102.10423 [cs]*, Feb. 2021, arXiv: 2102.10423. [Online]. Available: http://arxiv.org/abs/2102.10423

[55] Z. Chen, C. H. Yu, T. Morris, J. Tuyls, Y.-H. Lai, J. Roesch, E. Delaye, V. Sharma, and Y. Wang, "Bring Your Own Codegen to Deep Learning Compiler," *arXiv:2105.03215 [cs]*, May 2021, arXiv: 2105.03215. [Online]. Available: http://arxiv.org/abs/2105.03215

[56] M. Steuwer, T. Koehler, B. Köpcke, and F. Pizzuti, "RISE & Shine: Language-Oriented Compiler Design," *arXiv:2201.03611 [cs]*, Jan. 2022, arXiv: 2201.03611. [Online]. Available: http://arxiv.org/abs/2201.03611

[57] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010, conference Name: Computing in Science Engineering.

[58] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised Translation of Programming Languages," *arXiv:2006.03511 [cs]*, Sep. 2020, arXiv: 2006.03511. [Online]. Available: http://arxiv.org/abs/2006.03511

[59] "R-Stream Compiler." [Online]. Available: https://dev.reservoir.com/publication/r-stream-compiler-2/

[60] L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: https://doi.org/10.1109/99.660313

[61] H. Zhou and C. Liu, "Task mapping in heterogeneous embedded systems for fast completion time," in *2014 International Conference on Embedded Software (EMSOFT)*, Oct. 2014, pp. 1–10.

[62] C. P. Gomes and B. Selman, "Algorithm portfolios," *Artificial Intelligence*, vol. 126, no. 1, pp. 43–62, Feb. 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370200000813

[63] Z. Chen and B. Liu, *Lifelong Machine Learning: Second Edition*. Morgan & Claypool Publishers, Aug. 2018, google-Books-ID: JQ5pDwAAQBAJ.

[64] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54–71, May 2019.

[65] M. Brčić and D. Kalpić, "Combinatorial testing in software projects," in *2012 Proceedings of the 35th International Convention MIPRO*, May 2012, pp. 1508–1513.

[66] W. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 Dwarfs: A work in progress," *ICPE'12 - Proceedings of the 3rd Joint WOSP/SIPEW International Conference on Performance Engineering*, Apr. 2012.

[67] B. Johnston and J. Milthorpe, "Dwarfs on Accelerators: Enhancing OpenCL Benchmarking for Heterogeneous Computing Architectures," *Proceedings of the 47th International Conference on Parallel Processing Companion*, pp. 1–10, Aug. 2018, arXiv: 1805.03841. [Online]. Available: http://arxiv.org/abs/1805.03841

[68] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks," *arXiv:1810.02244 [cs, stat]*, Nov. 2021, arXiv: 1810.02244. [Online]. Available: http://arxiv.org/abs/1810.02244

[69] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '21. New York, NY, USA: Association for Computing Machinery, Mar. 2021, pp. 610–623. [Online]. Available: https://doi.org/10.1145/3442188.3445922

[70] D. Patterson, J. Gonzalez, U. Hölzle, Q. H. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink," Mar. 2022, publisher: TechRxiv.