Memory Utilization and Machine Learning Techniques for Compiler Optimization

Shreyas Madhav A V[1], *Siddarth* Singaravel [1], and *Karmel* A[1,*]

[1]School of Computer Science and Engineering, Vellore Institute of Technology, Chennai, 600127, Tamilnadu, India.

Abstract. Compiler optimization techniques allow developers to achieve peak performance with low-cost hardware and are of prime importance in the field of efficient computing strategies. The realm of compiler suites that possess and apply efficient optimization methods provide a wide array of beneficial attributes that help programs execute efficiently with low execution time and minimal memory utilization. Different compilers provide a certain degree of optimization possibilities and applying the appropriate optimization strategies to complex programs can have a significant impact on the overall performance of the system. This paper discusses methods of compiler optimization and covers significant advances in compiler optimization techniques that have been established over the years. This article aims to provide an overall survey of the cache optimization methods, multi memory allocation features and explore the scope of machine learning in compiler optimization to attain a sustainable computing experience for the developer and user.

1 Introduction

With the advent of modern computing technology, the scale and complexity of applications have increased vastly. This rapid expansion, in the scale in which applications are built, has given rise to a dire need for making maximum use of the computing resources present. The effectiveness and impact of any optimization technique is evaluated based upon the environment of compilation, architecture and application in which it is made use of. The selection and employment of the relevant optimization strategy plays a vital role in achieving maximum performance and requires the developer to have a good understanding of the system background. Proper optimizations lead to a drastic reduction in the cost of a product and help enhance it, without relying on higher speed hardware. This paper explores various optimization techniques based on memory allocation, cache-based optimization, ML based optimization in detail, to establish an overview of the current state of the technology. In order to work on optimizing compilers for better performance, it is essential for a programmer to understand the basic functioning and nature of a general compiler.

A Compiler translates high level language (source code) written by humans into low level language (machine code) by passing through various compilation phases such as Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, Optimization and Code Generation. While the properties of different compilers provide different variations in their structure and architecture, these steps provide a more generalized working of a compiler. The lexical analyser (scanner) converts source code, which is a stream of characters, into a stream of tokens (object that describes a lexeme). This involves removing comments, checking indentation and removing whitespaces. Now the code is split into lexemes and thereby a sequence of tokens is created. This is the reason why lexical analysis is also known as tokenization. Scanners need to be concerned with issues such as case sensitivity and whether comments are nested, or this might lead to lexical errors.

Now the sequence of tokens generated by the scanner phase is used to create a structure called abstract syntax tree (AST) which is a tree that depicts the logical structure of a program. This Syntax Analysis is also called Parsing and the compiler checks for syntax error as well as the grammatical structure. Each node of the AST is stored as an object and has no cycles. This AST will be used by the next compiler phase – semantic analysis. In this phase, the compiler uses AST to detect any semantic (meaningful) errors, such as performing operations on an undeclared variable or using the same name for multiple variables. The pieces/nodes of the syntax tree are tied to form a semantic graph which clearly describes the values of its attributes. This analysis consists of three steps which are type checking, flow control and label checking. Type checking involves checking type match in functions and method calls, flow control checking discusses flow control structure and label checking validates usage of labels. The legality rules differ, according to the chosen language.

*Corresponding author: ~~karmel.a@vit.ac.in~~

After the completion of the above mentioned processes, the compiler will generate one or more intermediate forms which is essentially a flow graph made up of tuples. This can be considered as some intermediate representation (IR) or as a program for an abstract machine. This IR must be easy to generate and converse into target code. Intermediate code is machine independent so there is very little need to consider differences in compiler properties and optimization techniques are more viable to be applied to this code rather than the machine translated code. This phase discusses efficiency of code following the three important rules. Firstly, the final code shouldn't change the meaning of the original program. Secondly, speed up time should be more and shouldn't impact the overall time. Thirdly, it should focus on consuming fewer resources and less memory.

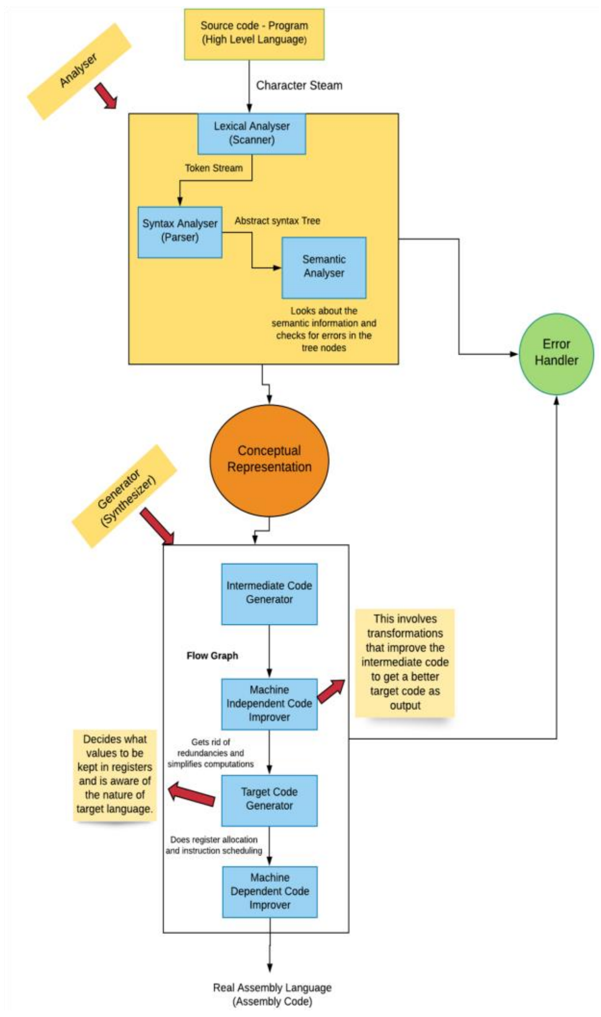
Source code - Program (High Level Language)
Analyser
Character Steam
Lexical Analyser (Scanner)
Token Stream
Syntax Analyser (Parser)
Abstract syntax Tree
Semantic Analyser
Looks about the semantic information and checks for errors in the tree nodes
Error Handler
Conceptual Representation
Generator (Synthesizer)
Intermediate Code Generator
Flow Graph
Machine Independent Code Improver
This involves transformations that improve the intermediate code to get a better target code as output
Gets rid of redundancies and simplifies computations
Decides what values to be kept in registers and is aware of the nature of target language.
Target Code Generator
Does register allocation and instruction scheduling
Machine Dependent Code Improver
Real Assembly Language (Assembly Code)

Fig. 1. Compilation Process

Optimization is classified into machine-dependent and machine independent techniques. In Machine Independent Optimization, the compiler accepts the IC (Intermediate code) and transforms it without any specific characteristic necessities required of the CPU registers and memory locations. The latter works on the generated target code according to the machine architecture involving CPU registers and have memory references. Optimization plays a crucial part over loops to save memory and CPU cycles. Loop optimization is done by moving the invariant code out of the loop and using cheaper expressions that reduces more CPU cycles without compromising the output.

This last compiler phase converts the optimized intermediate code to the machine code for that target machine. The final code should be efficient in such a way that memory and resource usage has been taken care of. After the assembly has been produced and we have the **.**asm file. This is then passed through an assembler and would generate the equivalent in binary. This binary code is written into a new file as an object (.o) file. Object files are machine code that are not executable and they need to be linked together to make it executable. Linker will make it executable as these are utility programs that compile the object code your backend generates.

2 Compiler Optimization

The main objective of compiler optimization methods and analysis is to get rid of all forms of possible redundant or inefficient blocks present in a computer program. It provides the system with the ability to manage the limited resources available and to efficiently maximize the output performance. The characteristics of a significant number of optimization techniques are inherent to a specific compiler and do not exhibit a general standard of properties. Hence, it is imperative to attain a complete understanding of the architectural dependencies of an application before deploying an optimization strategy to the code. The accuracy in achieving the intended purpose of the program and maintaining proper relations in the code are of primary importance. Several optimization techniques have been proposed in the past for different compilers and structures. A general categorization of the methods can be done on the basis of the levels of operation. This results in the categories - machine dependency, architecture based dependency and architecture independency. This paper investigates upon current mainstream strategies namely-Multi access memory based allocation, cache memory based optimization, machine learning techniques used for compiler optimization.

3 Multiple Memory Access Allocation

Multiple memory allocation is a relatively new method of optimization that works based upon the storing and retrieving of every instruction, onto multiple registers. A major issue faced in the field of embedded system design is the massive quantity and size of the programs. This optimization technique is a step towards resolving this issue. One of the main benefits that is provided by this method, for microprocessors, is the reduction in size of the executable code. Several data mapping and allocation strategies have been incorporated in this domain and varied approaches have been proposed in the past, for this technique.

Neil Johnson et.al [1] provided identification techniques in their proposed work for converting available groups of single instructions (load and store) and encapsulating them into multiple access instructions (load and store). This conversion of several instructions into a single multiple memory access instructions could be done efficiently by their proposed algorithm. It also guided the allocation and storage of resources in a manner that is precise, which previously is attributed only to opportunistic optimization. The implementation of their Solve MMA algorithm with the Value State Dependence based Graph framework is quite similar to the Simple offset algorithm for Assignment. Their model was analysed using an ARM thumb processor as the primary example and achieved significant code size reduction. Moving forward in this domain, the analysis done on the compatibility for combination of several instructions can be carried out in a more obsessive manner to identify and exploit more combinational opportunities. General purpose computations on graphic processing units can be optimized with the usage of novel optimizing compilers[2] that mitigate challenges of effectively utilizing the memory hierarchy present in the GPU and management of the parallelism. A functionally correct naive GPU kernel function is accepted by the compiler without considering the degree of performance optimization present. The memory access patterns are derived by applying code analysis methodologies to generate the appropriate kernel invocation parameters and kernel optimization techniques. Partition camping is eliminated by address offset insertion or remapping thread blocks. Memory bandwidth is improved using memory coalescing and vectorization.

An innovative way for the exploitation of re-usage of available data was proposed, with the main objective of reducing the number of operations carried out on the memory storage banks. The proposed approach [3] focusses primarily on the Coarse Grained Reconfigurable Architectures (CGRAs), which make use of a local multi bank memory. Figure 2 illustrates the typical architecture of CGRAs.

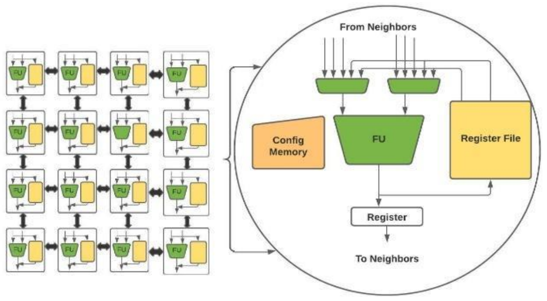

Fig. 2. CGRA Architecture

CGRAs are highly regarded for power efficiency and help reach peak performance metrics with optimal power. The entire architecture of the system is structured in a simple manner and the mapping of application holds the most complexity. The optimization technique used in this aims to solve the data mapping challenges faced in CGRAs where the same memory bank cannot be accessed simultaneously by two or more PEs. Their experimental results concluded their compilation approach that emphasizes on local memory awareness and generates mappings that provide a drastic increase in the performance when compared to typical scheduler does not focus on the memory allocation.

4 Cache Optimization Techniques

Making use of registers and cache memory to their maximum potential, in order to increase the efficiency of our computation is the core principle behind the process of cache optimization. The commutative and associative properties of the computations can be utilized to form a sequence of operations that retain the integrity of the results, while making best use of the memory resources available. The total number of instructions required can be reduced significantly by storing the frequently used data, close by. The use of an appropriate scheduling technique is also crucial in providing speedups to the performance. Activities such as reducing the hit time, increasing cache bandwidth, reducing the miss penalty, reducing the miss rate and reducing the miss penalty through parallelism.

A simple set of optimization techniques focussing on reuse of cache lines in order to find the favourable loop configurations was originally proposed by Steve Carr et al[4]. This ultimately optimizes the compiler by improving the data locality. The integration of loop fusion, reversal, permutation and distribution to provide an enhancement in data locality was one of the prime factors for the success of the optimization approach. This approach provides opportunities for improvements in machine independent programming, primarily in FORTRAN program optimizations. Cache locality optimization techniques focussed towards recursive programs have been developed in the past. Dynamic splicing and distinct function invocations by employing data effect annotations has helped recognize concurrency opportunities and reduction of reuse distance for faster data reuse access. Using non kernel lightweight threads, function invocations are executed in an interleaving manner and dependencies of a program are detected. Multicore execution is made possible by employing a nested fork join model.

With the expansion and rise of multithreaded architecture multithreading based solutions, adaptive optimizing compilations have come into existence that have extended optimization principles to multithreaded architectures. A data cache aware approach was proposed, that lowers the object conflicts misses by establishing a suitable data layout[5]. This approach serves as an expansion of the previously established cache based optimizations to multi-threaded architectures. CCDP application in single core architectures has been proven effective to achieve maximum performance. However the beneficial features are not exhibited when this method is employed in multi-architecture systems. The above mentioned approach has been a significant step forward in solving this problem by providing extensions to the existing CCDP. A general application scheduling approach as well as specific known application approaches have been demonstrated, employing the core principles of cache conscious data placement (CCDP). CCDP application in single core architectures has been proven effective to achieve maximum performance. However the beneficial features are not exhibited when this method is employed in multi-architecture systems. The above mentioned approach has been a significant step forward in solving this problem by providing extensions to the existing CCDP.

Another novel approach that aims to provide a combinational solution of both high level and low level optimization techniques to attain ideal results in multicore processors and parallel platforms[6]. Innovative approaches for parallel programs such as carrying out fundamental analytical strategies such as Side Effect as well as May-Happen-in-Parallel, combined with a Scalar Replacement method for eliminating the amount of load form the high level optimizations part of the proposed approach. Another major improvement in this paper is the introduction of a new Isolation consistency model that outperforms established models in terms of providing replacement transformation chances. Retaining the properties exhibited by Linear Scan, in terms of the time of compilation and memory efficiency forms the central idea of the novel approach that deals with allocation of registers. This is included in the low level optimization strategies part of the model. The experimental results of the research work considered the allocation phase on the Bipartite Liveness Graph as an optimization problem and the main motive of the assignment phase is the prevention of the total count of spill instructions. Data Locality based scheduling for an OpenCL compiler was implemented by intense analysis of the memory locations accessed in order to set up a proper scheduling strategy [7]. This procedure leads to the optimization in the order of memory access and its patterns. Previously established work did not take the relation between memory access patterns and work scheduling into account. The changes made to the system in this paper has a significant influence on the speedup and has been proven against standard benchmarks (Parboil and Rodinia).

5 Machine Learning Based Optimization Techniques

The growth of interest in the field of machine learning has increased exponentially, transforming the small computing domain to one of the world's most captivating fields in technology. The reason behind this rise in interest is due to the massive flexibility of the domain, with the ability to be employed in different applications, as a means of improving efficiency and accuracy of the original system. Machine learning is of prime interest in the field of compiler optimization as it has the capability to provide assistance to the programmer on the necessary code optimization techniques that need to be employed for maximum performance, in terms of both amount of power consumed and execution time. A machine learning model can help the programmer understand the best technique for optimization and also evaluate the existing techniques employed. The perspective of considering the finding of code improvement opportunities as an optimization problem has solidified the position of machine learning models as the provider of the optima. Compiler developers focus on minimizing execution time, reducing code footprint and the usage of minimal power as criteria to judge the effectiveness of changes made to the compiler heuristics and techniques applied for optimization. Machine learning algorithms can be configured and modified to make the appropriate decisions for the developers. This can be categorised into a two-step process. The first step constitutes the learning of the model by utilizing data provided for training. The second step is deployment of the model to the real time environment for predicting the optimization results for unknown programs. Compiler options are determined by the cost functions and compiler heuristics are extremely dependent on this. The quality metrics employed can differ based upon the optimizations goals. These metrics include execution time, energy consumption, code size etc. The implementation of a cost function is essential for the compiler to choose the best optimization option from a wide collection of techniques without running the program separately by incorporating each technique into compilation. Based on the efficiency score obtained from the model, we can decide upon whether we want to rectify the problem or not.

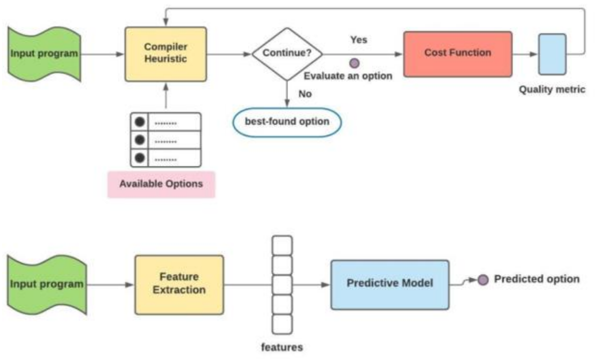

Fig. 3. Prediction Process

Decision trees have been the go to models for several optimization problems in the past [8]. Research conducted along these lines were geared towards identifying loop unroll factors[9], strategies for loop optimization and parallelization.[10], determining the viable and efficient algorithm implementations[11] and making decisions on GPU acceleration's profitability. The ability to interpret and visualize the process of decision trees makes it advantageous in multiple applications. Support Vector Machine[12] based classifications have been carried out in advanced cases for predicting optimizing opportunities[13].

Ensemble models involving SVMs[14], Decision Trees and K means Clustering were also tested out for this purpose and proved efficient in identifying optimization opportunities[15]. The machine learning model eases the burden of the programmer by automating the ideal optimization finding process. There is scope for research in determining whether the ML model can be extended to other compiler scenarios like analysis. With the rise of multicore architectures which provide parallel solutions to computationally intensive tasks, reinforcement learning based[16] and decision tree based[17] options for scheduling parallel sections of the programs are now gaining interest. The models have also been implemented for predicting the optimal number of threads[18] required for seamless execution of the program.

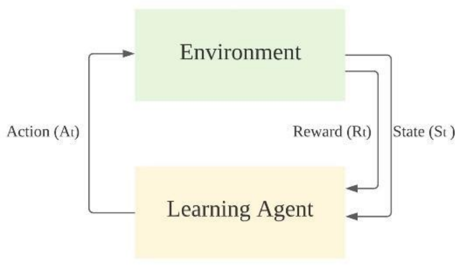

Fig. 4. Reinforcement Learning Model

The environment of execution plays a huge role in determining the effectiveness of an RL model. As the possible actions available to the model increases, the time to reach the convergence point to a proper solution also increases. The uncertainties in the environment in a dynamic computing system can be increased due to unpredictable disturbances. Today, RL algorithms have been combined with deep learning for learning of the value function in order to produce the most accurate results[19].

While static code feature extraction has no runtime overhead, it has its own set of vulnerabilities. The presence of data about code segments that are rarely executed can mislead the machine learning algorithm. Loop bound information can only be attained by the compiler during execution as it depends on the nature of user input. Resource content is not experienced in static code features and depends upon the available hardware. Hence dynamic compilers have been on the rise. The concept of ML has also been extended to the dynamic compiler by formulating a neuro-evolution based ANN network that predicts the order of benefits obtained when a code segment is being optimized [20]. Markov decision based phase ordering and current state code characterization are two of the main attributes of this approach, which is tested against standards set of Java benchmarks and shows significant increase in efficiency.

Extensive and exhaustive searching of compiler optimization techniques due to the large number of options present is infeasible on a large scale. This approach towards figuring out compiler optimizations is categorized as autotuning[21] or iterative compilation[22]. Iterative optimization techniques for compilers portray high efficiency and increased performance but are limited by the large cost of evaluations. A scalable approach, that is independent of compiler architecture, was implemented with the goal of increasing the speed of iterative optimization[23]. This novel approach reduces the number of evaluations using predictive modelling in order to locate the parts that are estimated to provide the maximum increase in performance. The model trains by modelling features and shape properties obtained by iterative evaluation, done on a defined set of programs. This model is implemented to figure out the focus points for optimization on a new set of programs. Such models have been proven to increase average speed up significantly. Attaining proper features is vital and time consuming, the generation of features automatically from the intermediate representation of the compiler [24]. Approaches towards solving this require huge grammar to be provided and work based upon greedy programming[25]. A powerful technique that has been explored limitedly is the implementation of Gaussian Processes for optimization [26].Considered one of the most renowned techniques before the advent of deep neural networks, this method is a robust way of approaching optimization problems when the training data is minimal[27]. The confidence interval provided can be used by the programmer to analyse the pros and cons of a specific implementation scenario.

6 Conclusion

Starting from the basics of a compiler to modern day compiler optimization techniques, The paper aims to provide an overall survey of the imperative optimization techniques that are of use today. The paper provides a brief overview of the properties of the memory based and machine learning techniques that have been employed for the purpose of compiler optimization. Multi memory allocation and cache based optimization techniques have been elaborated on in the category of memory based optimization techniques. Machine learning techniques for identification of potential opportunities of optimization have been provided with details of each approach surveyed. The extension of these techniques to different compilers and architectures have also been explored. Further research can look into the diverse possibilities of using machine learning algorithms to improve data locality and reuse to provide indirect optimizations to the compilers.

References

1. Johnson, Neil & Mycroft, Alan. (2004). Using Multiple Memory Access Instructions for Reducing Code Size. 2985. 265-280. 10.1007/978-3-540-24723-4_18.
2. Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU compiler for memory optimization and parallelism management. SIGPLAN Not. 45, 6 (June 2010), 86–97. DOI:https://doi.org/10.1145/1809028.1806606
3. Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. 2011. Memory access optimization in compilation for coarse-grained reconfigurable architectures. <i>ACM Trans. Des. Autom. Electron. Syst.</i> 16, 4, Article 42 (October 2011), 27 pages. DOI:https://doi.org/10.1145/2003695.2003702
4. Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler optimizations for improving

data locality. SIGPLAN Not. 29, 11 (November 1994), pp. 252-262.

5. Sarkar S., Tullsen D.M. (2008) Compiler Techniques for Reducing Data Cache Miss Rate on a Multithreaded Architecture. In: Stenström P., Dubois M., Katevenis M., Gupta R., Ungerer T. (eds) High Performance Embedded Architectures and Compilers. HiPEAC 2008. Lecture Notes in Computer Science, vol 4917. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-77560-7_24
6. Barik, Rajkishore. "Efficient optimization of memory accesses in parallel programs." (2010) Diss., Rice University. https://hdl.handle.net/1911/62060.
7. Hee-Seok Kim, Izzat El Hajj, John Stratton, Steven Lumetta and Wenmei Hwu. "Locality Centric Thread Scheduling for Bulk-synchronous Programming Models on CPU Architectures," in Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO '15), February 2015.
8. A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in Proc. Int. Conf. Artif. Intell. Methodol. Syst. Appl., 2002, pp. 41–50.
9. H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in Proc. 7th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO), Mar. 2009, pp. 81–91
10. H. Yu and L. Rauchwerger, "Adaptive reduction parallelization techniques," in Proc. 14th Int. Conf. Supercomput. (ICS), 2000, pp. 66–77.
11. Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in Proc. 36th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI), 2015, pp. 379–390.
12. Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in Proc. 14th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP), 2009, pp. 75–84.
13. P. Zhang, J. Fang, T. Tang, C. Yang, and Z. Wang, "Auto-tuning streamed applications on Intel Xeon Phi," in Proc. 32nd IEEE Int. Parallel Distrib. Process. Symp. (IPDPS), 2018.
14. Z. Wang, G. Tournavitis, B. Franke, and M. F. P. O'Boyle, "Integrating profile-driven parallelism detection and machine-learningbased mapping," ACM Trans. Archit. Code Optim., vol. 11, no. 1, p. 2, 2014.
15. M. K. Emani and M. O'Boyle, "Celebrating diversity: A mixture of experts approach for runtime mapping in dynamic environments," in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '15, 2015, pp. 499– 508.
16. M. K. Emani and M. O'Boyle, "Change detection based parallelism mapping: Exploiting offline models and online adaptation," in Languages and Compilers for Parallel Computing: 27th International Workshop (LCPC 2014), 2014, pp. 208–223.
17. Y. Zhang, M. Voss, and E. S. Rogers, "Runtime empirical selection of loop schedulers on hyperthreaded smps," in 19th IEEE International Parallel and Distributed Processing Symposium, ser. IPDPS '05, 2005.
18. Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '09, 2009, pp. 75–84
19. Y. Li, "Deep reinforcement learning: An overview," CoRR, 2017.
20. Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. In Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12). Association for Computing Machinery, New York, NY, USA, 147–162.

    DOI:https://doi.org/10.1145/2384616.2384628
21. K. Datta et al., "Stencil computation optimization and auto-tuning on state-ofthe-art multicore architectures," in Proc. ACM/IEEE Conf. Supercomput., Nov. 2008, pp. 1–12.
22. P. M. Knijnenburg, T. Kisuki, and M. F. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," J. Supercomput., vol. 24, no. 1, pp. 43–67, 2003.
23. F. Agakov *et al*., "Using machine learning to focus iterative optimization," *International Symposium on Code Generation and Optimization (CGO'06)*, New York, NY, USA, 2006, pp. 11 pp.-305, doi: 10.1109/CGO.2006.37.
24. M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund, "Practical aggregation of semantical program properties for machine learning based optimization," in Proc. Proc. Int. Conf. Compil. Archit. Synth. Embedded Syst. (CASES), 2010, pp. 197–206.
25. H. Leather, E. Bonilla, and M. O'Boyle, "Automatic feature generation for machine learning based optimizing compilation," in Proc. 7th Annu. IEEE/ACM Int. Symp. Code Generat. Optim. (CGO), Mar. 2009, pp. 81–91.
26. C. K. Williams and C. E. Rasmussen, "Gaussian processes for regression," in Advances in neural information processing systems, 1996, pp. 514–520.
27. C. E. Rasmussen and C. K. Williams, Gaussian processes for machine learning. MIT press Cambridge, 2006, vol. 1.