Autotuning Program Optimizations using Predictive Models

Shravan Kale
Computer Science Department
University of Oregon
Eugene, USA
shravank@cs.uoregon.edu

***Abstract*—Programmers use compilers to translate their source code to machine code and expect the compiler to make decisions regarding optimizations. While the compiler-only approach can provide a decent baseline performance, applications such as High-Performance Computing require extracting maximum performance for a hardware architecture. Traditionally, compilers have used hand-crafted heuristic models to make optimization decisions based on the compilation state of a program for benchmark architectures. Given today's diverse hardware architectures and programming models, manually constructing them is resource-intensive and time-consuming. An alternative is required that can offer automation and portability of models that can make optimization decisions.**

Machine Learning and Deep Learning techniques offer a promising alternative as they offer automatic construction and portability across architectures. They are implemented as predictive models to make decisions for the selection, ordering, and parameter values of optimizations. To build such models, we must first characterize programs using representation features and associate them with a search space of optimizations. This survey discusses the use of predictive models in automating the tuning of optimizations, categories of the various sources for generating representative features of programs, and the areas where predictive models are employed for tuning optimizations.

***Index Terms*—autotuning, predictive models, program representations, compiler optimizations**

I. Introduction

Compilers are programs in itself that translate other programs from a human-readable high-level programming language such as C and Fortran to a machine-readable low-level programming language such as assembly language. They produce binary executables, which are executed on a target machine of a given architecture. A compiler for the C programming language translates a program written in C code to machine code. In contrast, a Java compiler translates Java code into bytecode for execution on an instance of the Java Virtual Machine. The translation process needs to retain the original intention of the programmer, characterized by correctness while producing an efficient translation. The efficiency of a program is measured by its execution time, code size, or energy consumption. The translation process comprises internal methods, which include lexical analysis, syntax analysis, semantic analysis, code generation, and optimization.

The process of optimization produces significant benefits in improving the efficiency of a translated or compiled program. Compilers such as GCC [1] and LLVM's Clang [2] make various optimizations available in the different stages of compilation. Optimization options are available for the source code, the abstract syntax tree representation, the intermediate representations, and machine code representation. The target of optimizations can be the entire program or a specific part of the program, such as loops. Optimizations such as Constant Folding and Common Subexpression Elimination help reduce the number of instructions or code size by reducing computations. Optimizations such as loop unrolling transform loops such that their bodies are replicated several times. Listing 1 shows an example of a loop transformed using loop unrolling. Compilers are built with a suite of optimizations that are applied based on decisions made by a compiler or can be user-specified using compilation flags or pragma directives. Programs can be compiled without optimizations or by selecting a default pipeline of optimizations. For example, LLVM's Clang frontend provides pipelines such as -O0 for no optimizations, -O1 for basic optimizations, -O2 for moderate optimization, -03 for maximum optimizations without time constraints, -OFast for aggressive optimization, and -Os and -Oz for code size optimization.

```
//Non-unrolled loop
for(int i=0; i<N; i++){

    A[i] = B[i] * C[i]
}

//Unrolled loop factor(4)
for(int i=0; i<N; i+=4){

    A[i] = B[i] * C[i]
    A[i+1] = B[i+1] * C[i+1]
    A[i+2] = B[i+2] * C[i+2]
    A[i+3] = B[i+3] * C[i+3]

}
```

Listing 1. Loop Unrolling Example

While the optimization levels or pipelines are adequate to a certain degree, they can leave unutilized performance benefits on the table for a given machine architecture. The application

of optimizations is guided by a compiler's rule-based heuristic model consisting of a cost function developed by compiler writers or developers. The model infers decisions by evaluating a cost function over their weighted measurement. The cost functions that determine the application of loop unrolling include cost evaluation of loop control overhead, instruction cache utilization, register usage, memory access latency, number of loop iterations, and size of the loop body. The cost functions that need to be evaluated and their associated weights are obtained by a manual selection and tuning process. Their construction is based on the developer's experience or trial and error methods using analysis of standard benchmarks on a set of standard architectures. While handcrafted heuristic cost functions have been beneficial, they can be time-consuming and resource-intensive to build for a given architecture. The problem is further exacerbated as hardware development has constantly evolved, including the addition of general-purpose and domain-specific accelerators and their associated programming models. As such, there is a requirement for efficient and automated construction of portable cost functions that can guide the compiler to extract as many performance benefits as possible by improving the application of optimizations. Machine Learning (ML) or Deep Learning (DL) models are promising alternatives to cost functions as their training can be automated to learn a cost function for a given architecture.

This survey explores the area of using ML/DL models for automating the tuning of optimizations or autotuning of programs. Section II presents an overview of autotuning using ML/DL as predictive models and discusses the search space of optimization, the training and inference of models, the techniques, and the evaluation metrics of said models. Section III discusses the various methods of characterizing or representing programs usable by the models. The section discusses the different sources of representations, their construction, and their usage with the models. Section IV discusses the various areas of autotuning optimizations and their associated search spaces. While the Section II provides an overview of the area, Section III and Section IV serve the purpose of categorizing the various works in the respective sections.

II. Autotuning using predictive models

Autotuning is the process of automating the tuning or searching for optimal configurations or parameter values of optimizations that may apply to a program. All the possible configurations or parameter values of optimizations are defined as the search space of the optimizations. Autotuning is applied to find optimal program variants to maximize performance or efficiency for the diverse set of available hardware architectures. The performance of a program is characterized by its execution time, whereas its code size or energy consumption characterizes its efficiency. The automation of the task alleviates the burden on a programmer to manually tune their programs for a given hardware architecture. The autotuning process is achieved by selecting optimal program variants, compiler-guided tuning, or by tuning application-specific parameters. Hardware vendors provide libraries such as Intel's Math Kernel Library [3] and AMD's Optimizing CPU Libraries [4] for CPU programs, Nvidia's cuBLAS [5] and cuDNN [6] for GPU programs. They provide hand-tuned low-level subroutines for linear algebra and deep learning optimized for their respective hardware. When vendor-supported libraries are not available, libraries such as ATLAS [7] and PHiPAC [8] can be used to tune BLAS routines for a given hardware or FFTW [9] library for Fourier transforms. This survey focuses on using ML/DL techniques in compiler-guided tuning or iterative compilation and application-specific parameter tuning in the following sections. The subsections also discuss the search space of optimizations, the construction of the ML/DL models as predictive models, and their associated metrics.

1) Search Space of Optimizations: The search space of optimizations is the set of all possible combinations of optimizations that can be applied to a program. These optimizations are available in the form of compiler flags that define transformations of the program code, as alternate program variants, or as user-insertable pragma directives that define parallelization strategies. An autotuner enables the search of optimal configurations or parameters of these optimizations. The combination of optimizations creates inter-dependencies, which can have unique effects on the programs to which they are applied. The combination of optimizations gives rise to the optimization selection problem and the phase-ordering problem of applying optimizations. The parameters of an optimization itself can further increase the size of the search space that requires tuning.

The selection problem of optimization consists of the task of selecting which optimizations need to be enabled or disabled. This problem is further exacerbated when sequences or a pipeline of optimizations are applied to a program. The presence or absence of optimizations can positively or negatively impact a program's overall optimization. The phase-ordering problem of optimizations consists of selecting the order in which optimizations are applied to programs. The ordering is also constrained by the pre-requirements of specific optimizations, which require the application of program transformations or other optimizations.

The loop unrolling transformation is a critical optimization [10] method, the application of which is determined by the size of the loop body, the number of loop iterations, the register size, and the instruction cache size. Enabling the transformation can improve instruction level parallelism and reduce branching overheads but can also degrade the performance of the instruction cache utilization and register usage. The unroll factor of loop unrolling, if too small, can result in no performance benefits, and if too large, can result in a non-optimal code size. The transformation can also have secondary effects on other optimizations, resulting in degraded performance. Hence, a determination needs to be made on enabling or disabling this transformation, its order relative to other optimizations, and its unroll factor.

2) Predictive Models: Datasets and Feature Engineering: A collection of programs must be acquired to train ML/DL

models. The acquired programs should represent diverse computationally intensive workloads yet be representative of standard low-level computations or code kernels such as matrix-matrix multiplication. Such programs or code kernels are available within standardized benchmark suites or datasets created for performance evaluation of CPUs, GPUs, parallelization of programs, and embedded system platforms. The SPEC CPU [11] and SPAPT [12] benchmarks are used to train models used in optimizing CPU code kernels, whereas the Rodinia [13] benchmark is used to train models used in optimizing GPU code kernels. For parallel programs, the NAS Parallel [14] benchmarks are used to train the machine learning models, and PARSEC [15] and MiBench [16] are used for embedded platforms. The SPAPT benchmark was explicitly designed for automatic performance tuning of various categories of programs, including linear algebra kernels, linear solver kernels, stencil code kernels, and statistical computing kernels. The categories consist of kernels such as tensor matrix multiplication, Lower-Upper decomposition, 3-D stencil computation, and covariance computation.

Once a dataset of programs is acquired for training the ML/DL models, they must be characterized by a representation consumable by the model. Feature engineering is the process of constructing such a representation for every program. The features can be engineered from various sources for any given program. They can be generated from the compilation phase using static analysis without requiring program execution. They can also be generated during the runtime of a program or using dynamic analysis, but it may require multiple executions of a program. Static analysis features can be sourced from a program's abstract syntax tree presentation or the intermediate representation. In contrast, dynamic analysis features can be sourced from hardware counters during the execution of a program. The various representation strategies are further discussed in Section III

3) Predictive Models: Training and Inference of ML models: After selecting a method of generating program features, the next step is constructing a training dataset for the ML/DL models. The dataset must consist of tuples of the format $\langle X_i, Y_i \rangle$ where X_i represents the features of the ith variant of a program and Y_i can be a metric to be tuned or a label in the case of classification problems. Program variants are constructed by applying optimizations from their search space and then recording their corresponding metrics to be trained. Fig. 2 shows the system of training a predictive model. Program variants can also be generated using other autotuner frameworks such as Orio [17] that can tune and apply different optimizations to a program.

The selection of a training model is driven by objectives such as the program representation's structure and the training task's objective. Supervised learning uses models such as Support Vector Machines (SVM) [18], Decision Trees, and Artificial Neural Networks (ANN) [19] when a vector of features is available. Decision Trees [20] and their ensembles can produce rules to analyze the model's decisions. For graph-specific representations, Graph Neural Networks (GNN) [21], Graph Convolutional Neural Networks (GCNN) [22], and Graph Transformers [23] can be utilized to model programs. Models for unsupervised learning, such as K-Means and autoencoders, can be used to generate intermediate solutions or features. The autotuning problem can also be modeled as a Reinforcement Learning problem where models like ANNs can perform the role of a policy agent whose task is to select actions that produce an optimal reward.

Once a model is trained using the training dataset, it can be used for making predictions either in the iterative compilation setting or in the direct prediction setting. The construction of these models can have a higher initial cost in terms of time and computations. However, its utility is that they can be used for the prediction phase after the initial training phase with relatively cheaper inference costs. The features required for these models and their associated usage in tuning are described in the following sections. However, the architecture of the individual models is out of the scope of this survey.

4) Autotuning techniques: Iterative Compilation: The field of compilers also phrases compiler-guided tuning as iterative compilation. It is the process of a compiler that searches through the available set of program optimizations to find an optimal solution that minimizes metrics such as execution time. The search space consists of individual compiler optimizations or sequences of compiler optimizations. Fig.1 shows an example of the iterative compilation loop. The method of searching for an optimal parameter can be conducted using exhaustive search, random search, heuristics-based search, or predictive modeling. Exhaustive search can be cost-intensive and time-consuming as the entire search space needs to be evaluated. In contrast, random search is efficient but has no guarantees of finding the global optima of the search space. Exhaustive search is efficient only if the search space is tractable for autotuning in compilers; otherwise, random search is the pragmatic solution.

Heuristics-based search relies on a cost function to guide the progression in search; it can be an effective method but still requires evaluating optimizations in the search space. The heuristic-based search methods can be categorized as global optimizations that target exploration and exploitation of the search space or as local optimizations that target only the exploration of the nearest promising space [24]. Global optimizations include Simulated Annealing, Particle Swarm Optimizations, and Genetic Algorithms. In contrast, local optimizations include Nelder-Mean, Orthogonal Search, and Variable Neighborhood Search. Predictive modeling offers an efficient method by reducing the cost of repeated program evaluations to a one-time cost of training the model. Predictive models based on ML and DL have been found to be an ideal solution in iterative compilation. As shown in Fig. 1, the heuristic cost function can be replaced with a predictive model whose training phase is shown in Fig.2. Compilers such as GCC, LLVM's Clang, and Pluto [25] can guide the tuning process using heuristic models constructed by their respective developers. Compilers with predictive models have also been developed, as is the case with Halide [26] for image processing

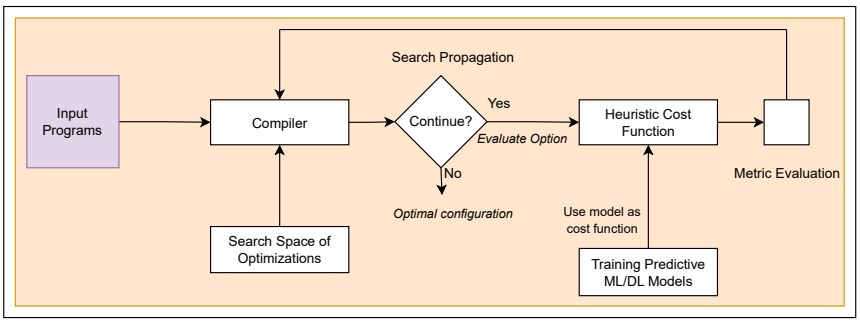


Fig. 1. Iterative Compilation Loop

pipelines, AutoTVM [27] for tensor computation programs, and Tiramisu [28] for CPU programs.

5) Autotuning techniques - Direct prediction: The process of direct prediction does not involve the predictive model as a part of the iterative compilation loop. The models are directly used to make predictions over the distribution of the input space of programs. It is useful when making tuning decisions over the application-specific parameters of a program. The parameters in the case of direct prediction can be any program parameter that can generate program variations. For example, in loop unrolling, the application of the transformations and its unroll factor form the parameters for autotuning. The search space for the tuning is defined as all the possible values of its parameters that can generate program variants.

Figure 2 shows an example of training a model with a direct prediction strategy. This strategy can be used for predicting unroll factors of loop unrolling [10], tile sizes for loop tiling [29], memory allocation and scheduling policies for OpenMP [30], parameters to compile CUDA programs for arbitrary input size of programs [31], and thread coarsening factors of OpenCL programs [32].

6) Metrics - Execution Time: Predicting the speedup of a program or part of a program is one of the most popular objectives of tuning compiler optimizations. The speedup objective is popular as it directly characterizes a popular use-case of optimizations, which is faster execution of programs on a given target hardware architecture. The speedup calculated is relative to a baseline such as the aggressive optimization sequence -03 used by compilers or the default pragma directive for OpenMP programs. In the case of loop optimizations, speedup can be relative to applying no optimizations or other configurations of loop optimizations in the case of code similarity. Speedups are calculated for making compiler pass selections, comparing the order of compiler pass sequences, tuning code similarity, and OpenMP parameter selection. The abovementioned examples are covered further in Section IV.

7) Metrics - Energy Consumption: Compilers such as GCC and the Clang/LLVM toolchain do not offer default pipelines for optimizing the energy consumption of a program as they do for code size and execution time. As such, it is generally left to the program developers to tune their programs to support architectures such as embedded systems where energy resources might be constrained. Writing energy-efficient HPC applications is also beneficial, as HPC systems in data centers may have assigned per-node power budgets to reduce the expenses of cooling solutions. As energy values are continuous real values, regression is the method of choice used for their prediction.

The power consumption of programs can be predicted on an instruction level using linear regression models [19]. The concurrency of parallel programs may not scale linearly, and hence, it can be throttled to improve energy efficiency. ANNs can be used to model programs using their power usage to make runtime decisions on concurrency levels and the placement of threads for a given phase of a parallel program [33]. For HPC applications, optimizing for energy consumption can be at odds with maximizing the performance of said applications. Joint-tuning metrics such as Energy-Delay Product [34] can account for power and performance. Power constraints such as Thermal Design Power caps can be included in the search space of optimizations as is done with tuning OpenMP parameters of parallel programs using a Graph Convolutional Neural Network [35].

8) Metrics - Code Size: Code size is used as a metric for gauging optimization tuning and is characterized by the size of the binary executable or the number of instructions. Optimizing the code size of a program also provides benefits in terms of execution time performance and energy consumption. Larger programs have relatively larger instructions to be processed, which can increase their execution time and increase cache misses, causing memory latency issues. Larger programs also consume more energy, which might be of concern on embedded platforms with power and data storage constraints. Optimizations such as Constant Folding and Common Subexpression Elimination target code size reduction by reducing the number of instructions. Compilers such as GCC and Clang/LLVM provide default optimization pipelines for code size compression using flags such as -Oz and -Os. The default pipelines are also used as baseline benchmarks for gauging the relative difference in code size.

The YaCoS [36] framework uses code size as a metric to predict optimization sequences using the MilePost-GCC compiler. The MLGo [37] framework replaces the heuristic-based inlining-for-size optimization in LLVM with a reinforcement

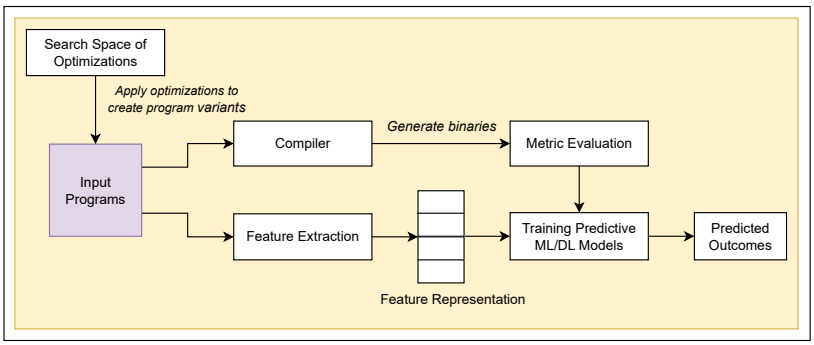


Fig. 2. Direct prediction of program optimizations

learning system. The reduction of code size depends on the effective enabling of code duplication optimization, which can also enable other optimizations. ANNs have been used to predict the code-size impact of code duplication on other optimizations [38].

III. Characterizing programs with Feature Representations

This section discusses the various sources of characterizing programs. Programs must be characterized by representative feature vectors or structures, such as graphs, to train the ML/DL based predictive models. The models predict objectives such as execution time or optimization parameter values. Each subsection details the feature extraction process and its usage with the predictive models.

A. Static Features - from the intermediate representation of compilers

As modern compilers produce intermediate representations to apply source code and hardware agnostic optimizations, the said representations are a rich ground to extract information to characterize a program. Compared to alternative approaches like the construction of graphs or polyhedra models, intermediate representations (IR) are readily available in modern compilers such as LLVM IR. Therefore, they are also cost-effective methods of generating feature vectors required for machine learning models. The features populating feature vectors can be function names, statistics of instruction types, and loop-related statistics. With features extracted from IR post-optimization, there is a correlation between the source code and the selection and ordering of the optimizations. The correlation can be exploited to create a machine learning model that learns a function mapping the correlation based on a given objective, such as the execution time speedup of a program. The generation of such features also aids a model in finding similarities between code kernels that can benefit from similar optimizations.

The extraction of feature vectors from the intermediate representations of programs can be akin to hand-crafted feature engineering commonly seen in training machine learning models. A set of features are described before their values are extracted from programs. The set of features, selection of subsets of features, and associated dimension reduction, if any, directly affect the performance of the machine learning models. Hence, the literature contains methodologies for feature selection based on the end objective of a machine learning model.

1) Generic user-defined static features: MilePost-GCC [39] embeds feature extraction into a compiler such as GCC constructed as a compiler pass. The first phase of the pass includes the generation of relational representation structurally different from LLVM-IR but includes information similar to the latter representation. The relational representation consists of entities such as types, variables, basic blocks, instructions, and the relationship between said entities. The second phase of the pass includes extracting vector features from the relational representation. These features consist of statistics based on the number of basic blocks that qualify additional conditions such as the number of predecessors and successors, number of edges and their types, number of instructions and their types, number of method calls, and occurrences and references of variables. A total of 56 such features are extracted that represent a generic characterization of a code kernel with its associated values. The Table I shows a sub-sample of features and their descriptions. The selection of the important features is left to be deduced by the machine learning model. Said features can be used independently for Probabilistic Machine Learning models or combined with combinations of optimizations to create a super-set for Transductive Machine Learning models.

2) Method-specific or loop-specific static features: Complementary to the approach of extracting generic features for representing programs, features are also extracted for specific tasks such as predicting loop unroll factors. Targeting specific optimizations requires features dedicated to parts of the source code, such as loops affected by said optimizations. The code kernels used in [10] are loops drawn from different benchmarks. The extracted features consist of loop-related features such as source language, trip counts, number of operations in a loop and their types, height of memory and control dependencies, and number of use-defs relationships in loops. Thirty-eight such features are collected initially but are further reduced to five using mutual information scores for optimal

TABLE I
SUB-SAMPLE OF FEATURES FROM MILEPOST-GCC [39]

Feature Category	Feature Description
Basic Blocks	Number of basic blocks in the method Number of basic blocks with a single predecessor Number of basic blocks with number of instructions in the interval [15,500]
Control Flow Graph	Number of edges in the control flow graph Number of critical edges in the control flow graph Number of abnormal edges in the control flow graph
Method-specific	Number of direct calls in the method Number of conditional branches in the method Number of instructions in the method
Variable and Constants	Number of local variables referred in the method Number of local variables that are pointers in the method Number of occurrences of integer constant zero

TABLE II
LOOP SPECIFIC FEATURES FROM [10]

Features
The loop nest level
The number of ops. in loop body
The number of floating point ops. in loop body
The number of branches in loop body
The number of memory ops. in loop body
The number of operands in loop body
The number of implicit instructions in loop body
The number of unique predicates in loop body
The estimated latency of the critical path of loop
The estimated cycle length of loop body
The language (C or Fortran)
The number of parallel "computations" in loop
The max. dependence height of computations
The max. height of memory dependencies of computations
The max. height of control dependencies of computations
The average dependence height of computations
The number of indirect references in loop body
The min. memory-to-memory loop-carried dependence
The number of memory-to-memory dependencies
The tripcount of the loop (-1 if unknown)
The number of uses in the loop
The number of defs. in the loop

performance of the modeling objective. The Table II lists the loop-specific features in detail. The extracted features are used to train ML models to predict the optimal unrolling factor for loops. The work in [20] also extracts integer-valued features based on categories such as memory access, arithmetic operation counts, loop body size, control statements in the loop, and number of iterations. The total features from all the categories are reduced to six using cross-validation. The features are used as input to decision trees to predict equal, improved, degraded, or insignificant performance improvement upon optimization application on loops of unseen programs.

Just as loops are targets of specific optimizations such as loop transformations, [40] shows that program method-specific optimizations can improve their performance over optimizations selected for the whole program. The method-specific features extracted from the program's bytecode include the code size of a method, the presence of exception handling in a method if the method contains no calls and method access specifiers such as private or public. Additional features include the distribution of instruction type categories in a method. The above features are combined with other runtime features to train a logistic regressor for the objective of optimization selection. The logistic regressor addresses the feature importance selection without any specific dimensionality reduction.

3) Multiple passes of methods to extract static features: Compilers apply optimization passes in a sequence based on the evaluation of their heuristic cost function. These evaluations are based on the updated intermediate representation from applying a previous pass(es). This behavior can be modeled by repeating the extraction of predictive model features after applying a subset of optimizations.

The GCC intermediate representation can represent programs [18]. The representations are available in a tree-like format using the Gimple or Generic language. Features are mined or extracted from the tree representation using templates that specify a pattern defining the features. The features are associated with the type of nodes in the tree representation; the nodes include loops, variables, call and condition expressions, and operators. The values for the features of a given program are extracted by traversing the tree representation using a GCC API plugin system with events and function callbacks. The features are further refined using genetic algorithms as a feature selection method. An SVM model is used to train with the features of a program with multiple sequences of optimizations. The feature values continually evolve upon applying a set of optimizations in a given sequence. Therefore, the model is also shown the evolving feature values during its training and prediction phases. Optimizations are grouped and divided by a set of milestone passes. The GCC-provided passes such as phiopt, sccp, forwprop, and vect are deemed milestone passes. Before every milestone pass, new feature values are extracted and used to train an SVM model, along with a new set of passes before the next milestone pass in a sequence. The model then learns in the training phase or predicts in the prediction phase of the SVM, to which optimization passes need to be executed based on the current set of feature values. Experiments in [18] show that extracting features based on a template in multiple phases improves compilation times of programs over extracting features just once using user-defined static features [39] or using GCC's aggressive -03 optimization.

The LLVM -03 pipeline consists of a series of optimization passes applied by a compiler after continuously evaluating the LLVM IR. The evaluation of the IR is conducted by using Multi-Layer Perceptrons (MLP) as predictive models after every subset of the LLVM -03 pipeline. The model proposed in [41] consumes the IR-related features after applying previously predicted passes of a subgroup to make predictions on the next subgroup of passes. A new set of features is extracted after applying a subgroup of passes. This method achieves

the optimal selection of passes in a given subgroup of passes. The features extracted for a program are static code features, which include counts of basic blocks, switch instructions, load/store instructions, direct calls to function definitions, and the maximum loop depth of functions.

B. Static Features - from structural representation like graph

Graph representations are presented as an alternative to static and dynamic representations of a program. A generic graph representation encodes information at its vertices and edges and from its neighborhood of vertices and edges. The neighborhoods of a vertex in a program representation can capture the dependencies or the relations between entities modeled as vertices and edges. Compared to the statistical representations of a program, graph representations can provide instructions with context relative to other instructions in a program. This context is introduced with the help of control-flow and data-flow dependencies of a program, which are not captured by statistical features. The dependencies are expressed via edges between the vertices, while the vertices can express entities such as variables, statements, and basic blocks of instructions. Unlike dynamic representations, graph representations are easier to collect as they do not require multiple program executions for a program's potential different input sizes. Compilers provide tools to generate graph representations such as Abstract Syntax Trees, Control Flow Graphs, Dominator trees, and the use-def relationships of variables.

1) Token Sequences and Abstract Syntax Tree-Based Representation: The raw source code of a program is not used to represent programs as it contains semantically irrelevant information such as custom identifier names and programmer idiosyncrasies and lacks dependencies of control or data flow. Semantically irrelevant information can be handled by normalizing a program. The process of normalizing [42] a program includes removing comments whitespaces and standardizing a program's identifiers. The normalized program's linear sequence of tokens is then converted to a fixed-size embedding representation using a language model. The language model consists of a sequence encoder, an embedding layer, and an LSTM model. The embedding representation of the program can then be combined with auxiliary inputs such as the input data size of the program or architectural parameters [43]. The combination of the embeddings and auxiliary input can be used to train a feed-forward network to predict on tasks such as heterogeneous device mapping or GPU thread coarsening.

A program's abstract syntax tree represents the hierarchical code structure and the syntactical relationships between code constructs such as variables, operators, expressions, and functions. The hierarchical structure consists of paths between its terminal nodes, which can be exploited to create embeddings representative of the source program. Code2Vec [44] is one such model that extracts these paths from a code snippet and represents them as a bag of vector representations. A weightage of the path vectors is learned using attention mechanisms in a neural network to aggregate into a single embedding

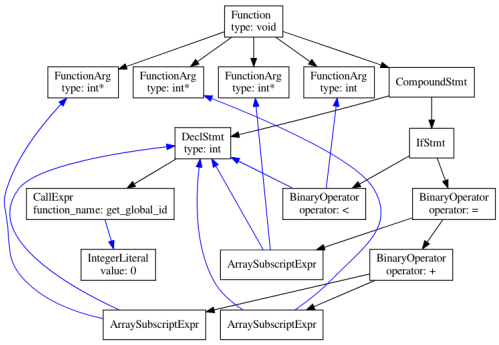


Fig. 3. Abstract Syntax Tree (AST) representation of a sample code with data-flow dependencies(blue arrows) [21]

representation relative to the label of the objective task used in its training. Though similar to Word2Vec, Code2Vec produces generalized embeddings of a program or code snippets that may be used for other tasks. The generalized embeddings can represent loops in a program to tune their vectorization in a reinforcement learning environment [45]. The policy agent learns a policy that maps the loop representation to optimal vectorization factors by inserting the appropriate Clang pragmas. It compiles the modified program using Clang to observe the execution time improvements as its reward function.

Programs represented by a sequence of tokens impose an absolute ordering, which may break commutative expressions such as addition [21]. In contrast, the AST of a program does not impose such an ordering as it abstracts away syntactically irrelevant structure. The structure of the AST can be used to represent programs, but it needs to be enhanced with dataflow dependencies between its variable identifiers. ASTs can be enhanced with edges representing the dataflow, such as nodes that refer to the same data [21]. The nodes of the AST digraph are labeled as declarations, statements, and types and values as defined in the Clang AST. The edges are labeled based on the kind of relationship in the AST, specifically child-of relationships and use-def relationships of variables. The AST structure for a sample code is shown in Fig.5. The digraph can be used directly with a message-passing graph neural network to predict the thread coarsening factor for OpenCL programs.

2) Graph-Based Representation: The Program Dependence Graph [46] representation predates the advent of machine learning but serves historical relevance in the representation of programs as graphs. The PDG structure encodes a program's data and control dependencies from data-flow and control-flow analysis, respectively. A Control Flow Graph (CFG) represents a program's control dependencies, which consists of vertices as basic blocks of instructions and edges that model the flow of control between the basic blocks. The data dependencies of a program are represented by edges of a Data Flow Graph, which consist of vertices as program statements or operators and operands. The PDG construction involves deriving a flow graph, which is an efficient approximation of the generated dependence subgraphs of the control flow and

data dependencies. A generic PDG consists of statements as vertices and edges as control and data dependencies. Variants of the PDG, such as a functional, operational, or hierarchical PDG, can be constructed by varying the representation level of statements for its set of vertices. The inclusion of the control and data dependencies in a graph representation enabled the expression of relationships between computationally related parts of a program. The expression of such relationships enables the application of program optimizations upon a single walk of said dependency-based graphs. Optimizations such as code motion require the interaction of both dependencies.

The control flow graphs (CFG) of a program can form the simplest graph representation of a program. However, the CFGs need to be modified as machine learning models can not consume basic block instructions. The nodes of a CFG can be represented with a feature vector where the vector encapsulates statistics of the instructions in basic blocks [47]. These statistics include the number of instructions, the number of load/store instructions, the number of Add/Sub/Mul/Div instructions, and the number of conditional branches. The feature vectors of all the nodes in a CFG are combined with the topological information or a list of directed edges of the CFG. The combined information is used to train an SVM model that supports a kernel function that can take as input structured data like a CFG. The SVM model is trained with the program representation and a set of loop optimization sequences to predict the speedup of using an optimization sequence. The work in [47] also compares the use of the dominator tree representation of a program but finds empirically the better performance of CFG as representations in its experiments.

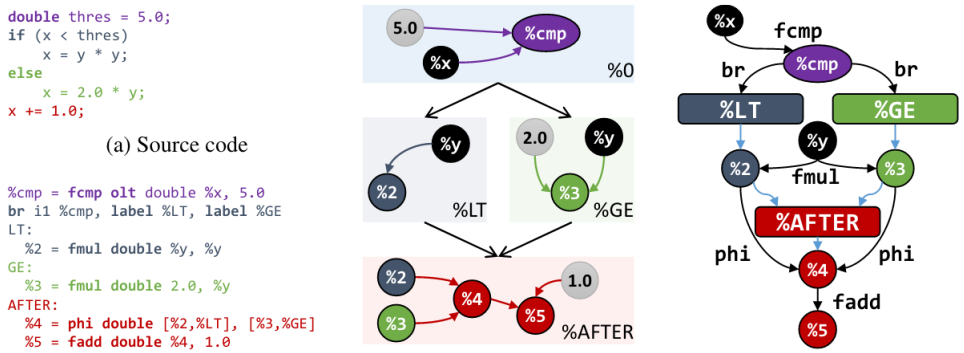
Contextual Flow Graphs (XFG) [48] are constructed as another representation suitable for deep learning models, whereas PDGs are used by compilers directly for optimization. XFGs encode data flow and control flow dependencies to create a unified representation. The XFG representation leverages the single static assignment of the LLVM-IR to track data flow between instructions or statements and the phi-expressions of the LLVM-IR to track the probable control flow of instructions. XFGs are directed multigraphs where the nodes can be variables or label identifiers such as basic blocks and function names. Edges of the XFG can represent the data dependence or control-flow dependence between the nodes. The Fig.4 shows the construction of the graph from a sample source program. The structure encodes paths for branches, loops, and functions. The construction of the XFG takes place over two passes of the LLVM-IR, and the dependencies are gradually inserted into the graph structure. The XFG representation also supports calls to external code using LLVM-IR of statically linked code, such as headers and static libraries or call statements for dynamically linked libraries. For the cases where the nodes of the XFG represent IR statements, the statements are converted to a vector representation in a continuous space using inst2vec [48]. The vector representations capture the semantics of the instructions in the XFG, whereas the XFG itself captures the structure of the code encoding data-flow and control-flow dependencies. The vector representations and the contextual information of the XFGs are provided as input to a Recurrent Neural Network to train it for various object tasks. These tasks include predicting hardware mapping for OpenCL programs given their code representation, input data size, and work-group size (number of threads in a group). Other objectives include predicting the optimal thread coarsening factor for OpenCL programs.

The embeddings generated by XFGs [48] and inst2vec capture the compiler IR's semantic information but are only used in downstream tasks. The generation of the embeddings or the representation of the program is not guided by the deep learning model, and hence, the embeddings may not be optimized for the downstream task [21]. Another way to generate a program representation is by constructing a digraph that represents a program's Control and Data Flow but is modified with calls and memory nodes [21]. The vertices are labeled with LLVM IR instructions, and the edges represent the data (as operator relationships with the IR) and control flow. Call edges are added to represent the calling relationship between functions. Edges are also added between memory locations, encapsulating load-store dependencies. The Fig.3 shows an example of the CDFG structure for an example source kernel. The digraphs are directly fed as input to a Graph Neural Network, which consists of the first layer that generates the initial embeddings from the nodes of the graphs a second layer that propagates the initial embedding and adds structural information from the edges of the graphs. A third layer aggregates the propagated embeddings and can additionally encode auxiliary input such as work-group size or data size for prediction objectives such as hardware offload mapping and optimal thread coarsening factor for OpenCL.

C. Dynamic Features - from performance counter measurement tools such as PAPI

Programs can be represented with static features extracted at compile time. However, static features do not characterize the run-time behavior of a program on a given target platform. Static features exploit the syntax of a program without knowing its run-time behavior. According to [49], static features that model code constructs such as loops are inefficient representations to characterize the run-time behavior of control-flow intensive large programs. An alternate representation system exists in the form of program counter values extracted at a program's run-time. The program counter values are measurements of pre-set events from registers in modern processors. These events can describe the characteristics of a program in terms of cache statistics, hardware cycle statistics, and branch instruction statistics. The measurements of these events have low overhead on the execution of the program and are thus ideal for representation extraction of program behavior at run-time. The tools used for the collection of these counter values include PAPI [50], Perfmon2 [51], and HPCToolkit [52].

The program counter values are used to populate a feature vector to represent a program for a given platform, which can be used to train machine learning models. However, the disadvantage of program counter values as representation is



(a) Source code (b) LLVM IR (c) Dataflow basic blocks (d) Contextual Flow Graph

Fig. 4. Graph Representation of Contextual Flow Graphs (XFG) The contextual flow graph is constructed from the intermediate representation (Fig. 4a) of a sample code (Fig.4b) with data-flow dependencies (Fig.4c) combined with control-flow dependencies (Fig.4d) [48]

TABLE III
PERFORMANCE COUNTERS OF PAPI [50] TO REPRESENT PROGRAMS IN [49]

Performance Counter Name	**Meaning**
HW_INT	Hardware interrupts
RES_STL	Cycles stalled on any resource
STL_ICY	Cycles with no instruction issue
TOT_CYC	Total cycles
TOT_INS	Instructions completed
VEC_INS	Vector/SIMD instructions
Floating Point Instruction Statistics	
FAD_INS, FML_INS, FP_INS, FP_OPS, FPU_IDL	Floating point: Adds, Multiplies, Total Insns, Total Ops, Cycles Idle
Branch Instruction Statistics	
BR_INS, BR_MSP, BR_TKN	Branch instructions, Cond. Branches Mispredicted, Cond. Branches Taken
Level 1 Cache Statistics	
DCA, DCH, DCM	Data Cache: Accesses, Hits, Misses
ICA, ICH, ICM, ICR	Instruction Cache: Accesses, Hits, Misses, Reads
LDM, STM	Load Misses, Store Misses
TCA, TCH, TCM	Total Cache: Accesses, Hits, Misses
Level 2 Cache Statistics	
DCA, DCH, DCM, DCR, DCW	Data Cache: Accesses, Hits, Misses, Reads, Writes
ICA, ICH, ICM	Instruction Cache: Accesses, Hits, Misses
LDM, STM	Load Misses, Store Misses
TCA, TCH, TCM	Total Cache: Accesses, Hits, Misses
TLB Statistics	
TLB_DM	Data translation lookaside buffer misses
TLB_IM	Instruction translation lookaside buffer misses
TLB_TL	Total translation lookaside buffer misses

that they are target platform architecture-dependent and do not offer the portability of optimization decisions.

The utility of performance counter values for program representation is proven by [49] using an example of a 181.mcf program from the SPEC CPU [11] benchmark. They compare the speedup and optimizations predicted by their performance counter-based model (PCModel) against the -OFast aggressive optimization scheme of the PathScale compiler. Unlike the aggressive optimization scheme, the comparison shows that the PCModel further exploits the run-time knowledge of L1 and L2 cache misses with the tested benchmark program. Even though both optimization schemes enable loop optimizations that improve locality, only the PCModel enables the generation of 32-bit code instead of the default 64-bit code, which aids in an improved execution time. The additional optimization is enabled due to the higher data cache accesses and a higher number of branch instructions than the average of the entire benchmark.

The PCModel uses logistic regression to map the performance counter values to a set of optimizations. The performance counter values are extracted from multiplexing special registers to generate values for 60 counters from program execution. These counters include the number of hardware interrupts and cycles, floating-point instruction statistics, branch instruction statistics, Level 1 and Level 2 cache statistics, and translation lookaside buffer misses. Table III shows a detailed explanation of the individual counters. The counter values are normalized by the number of instructions to generalize across

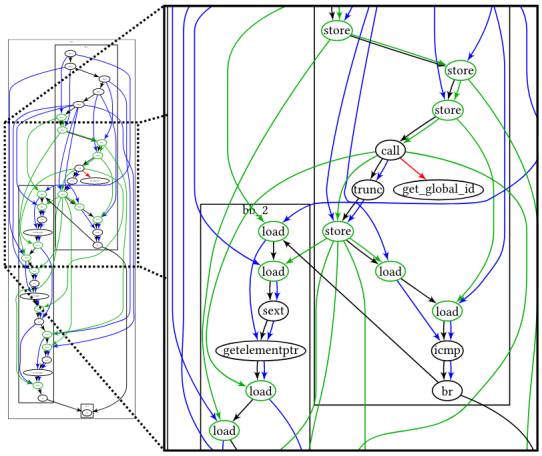


Fig. 5. Enhanced Control-Data Flow Graph (CDFG) of a sample program. The black arrows represent control-flow dependencies, the green-arrow represent data-flow dependencies, and the red arrows represent an external function call. [21]

different benchmarks. The model training set is constructed by randomly sampling 500 of the available program optimizations and applying them to the training programs. The speedups over an aggressive optimization scheme are evaluated to filter out sub-optimal optimizations. The feature vector of program counter values is used to train the logistic regression model that predicts a probability distribution over the filtered set of optimizations. The probability value of an optimization is used to determine the usage of an optimization for a given program.

Performance counters can also be combined with compiler optimization sequences to represent a program [53]. The program counter values extracted using PAPI in [53] include counter values the same as [49]. The optimization sequences are randomly sampled from Open64 compiler optimizations, and their speedup is recorded relative to Open64's -OFast optimization set. The sequences are represented using a bit vector where every element represents one of N possible values for a given optimization. The combined feature set of performance counter values and optimization sets is used to predict speedup for unknown kernels called a speedup predictor. The counter values can also be combined with two different sets of optimizations to predict a selection decision between the sets, called a tournament predictor. The counter values can also be used to predict the binary usage decision of a single optimization called sequence predictor. All three predictors are modeled using linear regression or support vector machines.

Dynamic features such as loop iteration counts, L1 cache misses, and branch miss rates can be extracted for sequential variants of a program. They can be combined with run-time metrics such as execution time for parallel variants of the same program. The combination of these features and static code features is used to predict the optimal number of threads and a scheduling policy type for parallel programs [54]. The extraction of the dynamic features requires the profiling of a program and may require multiple profiles for different input data. The requirement of multiple executions for profiling is dependent on the program itself. For a benchmark in [54], it was found that sequential variants of a benchmark program were sensitive to the input data as opposed to the parallel variant of the same program.

D. Dynamic Features - from measurements that are architecture agnostic

Performance counters dominate as a method for generating architecture-dependent representations. The performance counter's values are events extracted from special registers in modern processors. As such, these counter values as program representations depend on the underlying architecture, and the program representations are not portable across architectures. Target architectures can have variations such as cache sizes, buffer sizes, and number of registers. Representations independent of the underlying architecture are required to characterize programs if they need to be optimized across various target architectures. Such representations can be extracted by instrumenting a program at run-time.

The instrumentation of programs requires the insertion of extra code to profile program behaviors. PIN [55] is a portable software system that performs run-time instrumentation. PIN is expressed as a set of APIs used by tools such as MICA [56] to insert arbitrary calls to instrumentation or analysis routines during the execution of a program binary. It can also attach and detach as a process like a debugger to profile binaries with overheads limited to its attached time. MICA provides access to architecture-independent characteristics using tools such as PIN, ATOM, Valgrind, and DynamoRIO. MICA proved the viability of architecture-independent characteristics over architecture-dependent characteristics to represent programs by comparing the gzip-graphic benchmark of the SPEC-cpu200 [11] and the fasta benchmark of BioInfomark [57]. In its comparison, it finds that the values of the characteristics for benchmarks are similar in the case of architecture-dependent characteristics and different in the case of architecture-independent characteristics, thus asserting the benefit of architecture-independent characteristics over their dependent counterparts. The architecture-independent characteristic categories include register traffic, working set-size for instruction and data streams, global and local strides of data streams, and branch predictability. Table IV shows a detailed description of the features. The values for these characteristics are expressed in constants, probabilities, and percentages depending on the given characteristic. These characteristics are independent of the architecture but not the instruction set architecture, such as x86_64 and a compiler. According to [58], the MICA characteristics provide a sufficient high-level abstraction for the programs to be executed and their representations to be extracted on an x86 architecture but be used as optimized programs on a different target architecture.

Characteristics from profiling frameworks such as MICA are used as dynamic representations of programs to solve compiler optimizations' phase ordering problem [58]. The framework

TABLE IV
PERFORMANCE COUNTERS OF MICA [56] TO REPRESENT PROGRAMS

Characteristic Category	Measurement	Description
Instruction mix	6 percentages	Percentage of loads, stores, branches, arithmetic operations, multiplies, and floating-point operations
Instruction-level parallelism (ILP)	4 values	IPC achievable for an idealized out-of-order processor (with perfect caches and branch predictor) for window sizes of 32, 64, 128, and 256 in-flight instructions.
Register traffic	2 values and 7 probabilities	Average number of register input operands per instruction, average number of register reads per register write, distribution (measured in buckets) of register dependency distance, or number of instructions between production and consumption of a register instance.
Working-set size	4 numbers	Number of unique 32-byte blocks and 4-Kbyte memory pages touched for both instruction and data streams.
Data stream strides	20 probabilities	Distribution, measured in buckets, of global and local strides. Global stride is the difference in memory addresses between two consecutive memory accesses local stride is restricted to two consecutive memory accesses by the same static instruction. Strides are measured separately for memory reads and writes.
Branch predictability	4 percentages	Branch prediction accuracy for the theoretical prediction-by-partial-matching (PPM) predictor.8 We considered global and local history predictors, and per-address and global predictors.

MICA provides a set of 99 representations, many of which are correlated. To extract uncorrelated representations required for machine learning models, Principal Component Analysis is applied to the extracted representations. Programs are compiled with a pre-selected set of optimization sequences for which their representations are extracted using MICA and associated with their speedup values in a set of collected instances. The optimization sequences are a vector of combinations of a category of optimizations, including transformations over the dominator tree, control flow graphs, memory dependency, and loops. The previously collected instances are combined with new candidate optimizations (including repeated optimizations) and their associated speedup values over the program's older instance, forming a training set for a machine learning model. A linear regression model is used to learn the speedups for the new candidate optimization of a given program. The model is then used for predicting speedups for unknown programs where the input to the model consists of representations of the said program with a candidate optimization. The same program representations can also be used with a Bayesian Network to solve the phase ordering problem of compiler optimizations [59]. The benefit of using a Bayesian network is that an application-specific probability distribution can be used from a trained Bayesian network to guide an efficient selection of the candidate optimizations in the search space.

Another tool used to characterize the performance of programs is the Architecture-Independent Workload Characterization (AIWC) [60] tool. AIWC is used to instrument OpenCL kernels during the simulation of OpenCL program execution by the Oclgrind simulator. The tool can characterize a kernel's parallelism, its computational complexity, and its memory and control footprint in an architecture-independent method. The simulator generates notification events during program execution, which AIWC handles by populating data structures for each characterization. Compared to AIWC, MICA does not characterize the parallelism of an analyzed program. For AIWC, there exists a one-time cost of instrumentation of larger applications on a given simulator. Table V contains a list of feature representations and their associated descriptions. These representations are extracted for an OpenCL kernel on various hardware devices, including CPUs from Intel and AMD, and GPUs from Nvidia and AMD. The representations can then be used as an input to a Random Forest model to predict the execution time on a given hardware [61]. The collected metrics have a secondary benefit in that optimal metrics for a kernel can be inserted into its source code to guide a task scheduler on heterogeneous architectures.

E. Architecture-based and Polyhedral Features

1) Architecture Parameters as Representations: Compiler optimizations rely on carefully designed heuristics based on the abstraction of the underlying microarchitecture. As such, the architectural parameters can play an essential role in the effects of the interactivity of multiple optimizations. These parameters can be modeled by building architecture-sensitive empirical models [62]. The microarchitectural parameters can include the number of registers, the number of functional units, and buffer and cache sizes. Programs can be represented using features such as size, latency, and associativity of the data cache and l2 cache. Other features include issue width, branch predictor size, and register update unit size. These features can be combined with compiler optimization flags and their associated heuristics. Heuristics such as the maximum number of instructions in the callee and the maximum permissible increase in code size govern function inlining. The combination of these features can be used to predict the execution time of a given program relative to the underlying microarchitecture. A secondary benefit of constructing microarchitecture-sensitive empirical models is their ability to be used to explore their input search space. With a trained model, the optimal compiler optimization settings and heuristics can be searched for a given architecture using a genetic algorithm.

TABLE V
AIWC [60] TOOL METRICS TO REPRESENT PROGRAMS

Type	Metric	Description
Compute	opcode	# of unique opcodes required to cover 90% of dynamic instructions
Compute	Total Instruction Count	Total # of instructions executed
Parallelism	Work-items	# of work-items or threads executed
Parallelism	Total Barriers Hit	maximum # of instructions executed until a barrier
Parallelism	Min ITB	minimum # of instructions executed until a barrier
Parallelism	Max ITB	maximum # of instructions executed until a barrier
Parallelism	Median ITB	median # of instructions executed until a barrier
Parallelism	Max SIMD Width	maximum number of data items operated on during an instruction
Parallelism	Mean SIMD Width	maximum number of data items operated on during an instruction
Parallelism	SD SIMD Width	standard deviation across the number of data items affected
Memory	Total Memory Footprint	# of unique memory addresses accessed
Memory	90% Memory Footprint	# of unique memory addresses that cover 90% of memory accesses
Memory	Global Memory Address Entropy	measure of the randomness of memory addresses
Memory	Local Memory Address Entropy	measure of the spatial locality of memory addresses
Control	Total Unique Branch Instructions	# unique branch instructions
Control	90% Branch Instructions	# unique branch instructions that cover 90% of branch instructions
Control	Yokota Branch Entropy	branch history entropy using Shannon's information entropy
Control	Average Linear Branch Entropy	branch history entropy score using the average linear branch entropy

2) Polyhedral Representations: Programs can also be represented by mathematical relations, such as Pressburger Relations, as is done in Polyhedral Optimization frameworks. The mathematical relations or polyhedral representation of a program defines the loop constructs, which are targets of program optimization using loop transformations. Loops, including possibly imperfectly nested loops that qualify for polyhedral optimization, are called static control parts (SCoP). A SCoP representation consists of statement instances, the domain of the loop iterations, array access patterns and dependency analysis of the loop, and a multidimensional schedule that describes the execution of statements. The schedule of a SCoP can be modified to introduce loop optimizations such as loop interchange and loop fusion. The schedules can be modified so that multiple loop transformations can be composed to generate a new schedule.

Polyhedral representations can be combined [63] with existing program representations to create machine learning models that can predict the speedup of a program on the application of loop transformations. Performance counters can be combined with sequences of high-level optimization primitives such as loop tilling or parallelization. The primitives are encoded as a fixed length vector of bits called T. Each value of T represents the binary or numerical variations of the primitives, such as tilling enabled/disabled or the factors of loop unrolling. Multiple T vectors are generated for a given program P, which represents the application of the combination of various loop transformations. The loop transformations considered include loop tiling, loop fusion, thread-level parallelization, and SIMD-level parallelization. Programs are represented by a feature vector F, consisting of performance counter values from PAPI, including counters related to L1 and L2 cache, branch-related counters, translation lookaside buffer counters, and SIMD instruction counters. For each Program P, a vector of T optimization primitives are applied, and their speedups are recorded relative to the original program P. A speedup predictor model is constructed by training an SVM with the combination of the program's feature vector F and optimization primitives T. The model is used to predict the speedup by optimization primitives on unseen programs.

IV. AREAS OF TUNING OPTIMIZATIONS

The following section details the various areas where predictive models are employed for autotuning program optimizations. The search space of optimizations also needs to be defined to train ML/DL-based predictive models. The dataset for training the models is built by applying the optimizations from the search space on a program followed by its feature extraction. As such, the section also describes the search space for the given areas.

A. Clustering

Clustering is a method of grouping a set of entities represented by variables in order to assign the entity a membership to a group. A similarity metric such as the Euclidean distance or Hamming distance between its representation is used to assign an entity to a given group or cluster. The placing of an entity in a cluster also allows its categorization by its membership identifier and the inheritance of properties of a given membership. Clusters without a membership identifier are formed in unsupervised machine learning applications and with a known membership identifier in supervised machine learning applications. Programs can be modeled as entities with representations from Section III and membership can be modeled as a set of applicable optimizations. Clustering a set of programs can aid with reducing the search space of selecting optimizations. For the optimization of a program, compilers offer a host of optimization passes that can be selected and ordered before the application on the said program. It can be infeasible to evaluate all the possible combinations and permutations. Hence, there is a requirement for Design of Space Exploration technique to include the reduction of the

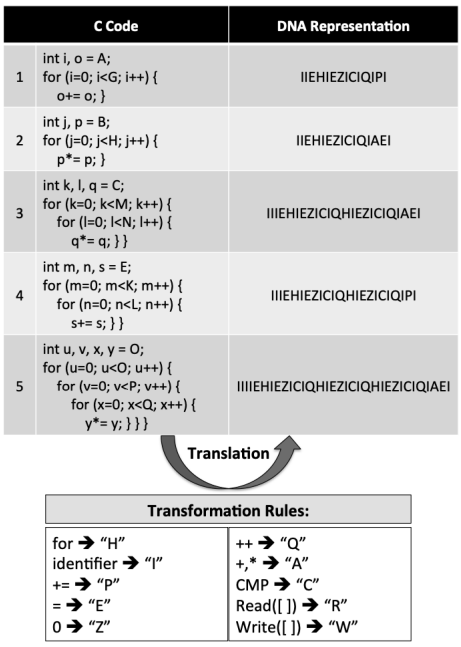


Fig. 6. The transformation rules used in [64] to convert the source code of a program to a DNA like sequence [65]

search space. As such clustering becomes an intermediate stage before other objectives are pursued.

Programs or functions within a program can be clustered into distinct groups with their associated optimizations passes [64]. The source code of programs can be converted into symbolic representations like a DNA sequence. The transformation rules for constructing such a sequence are shown in Fig.6. A distance matrix is calculated over the symbolic representation using Normalized Compression Distance [66]. A tree topology is constructed from the distance matrix using a phylogenetic reconstruction algorithm called Neighbour Joining Algorithm [67]. The clustering of program functions can be generated from the tree topology using a hierarchical clustering algorithm. An unknown program function can be matched with existing functions in clusters. The optimization passes in the matched cluster become the reduced search space of the unknown program function.

The need to reduce the search space of optimizations is also applicable to ordering optimizations in a sequence that generates its own search space, like that of the selection of optimizations. A reduced search space for the ordering of optimizations can be achieved by clustering optimizations independent of a program's representation. The -03 optimization sequence of the Clang front end of the LLVM compiler toolchain is a standard sequence obtained on a set of benchmarks. A tuning opportunity exists by exploring the search space of the selection and ordering of the optimizations. A dependence graph can be constructed that encodes via its edges the strength of the required or promising sub-sequences of optimizations in the -03 sequence [68]. An example of the graph is shown in Fig.7. The nodes of the graph represent the optimizations themselves. The weighted adjacency matrix of the said graph can be used by agglomerative clustering to generate clusters of optimization sequences. The clustering of the LLVM -O3 pipeline is shown in Fig.9

The clustering of programs can also be used for tuning optimizations modeled as a Recommendation System (RS) [69]. An RS system consists of users, items, and features of items that can be used to produce a "recommendation" based on a relevancy score. Programs can be modeled as users, optimization sequences as items, and the features of optimization as program representations upon applying a given optimization. The objective is to recommend an optimization sequence for an unknown program. RS systems consist of algorithms such as Content-Based Filtering (CBF) and Collaborative Filtering (CF) that can generate recommendations for optimization sequences. The distinction between the algorithms is the entity that is clustered. In the case of CBF, recommendations are based on the similarity of features of a program upon the application of an optimization sequence. Whereas in CF, recommendations are based on the similarity of programs independent of applying any optimization sequences. The metric used to make a recommendation is a relevancy score, a function of the similarity metric between the features and the programs.

B. Optimization Sequences

Compilers provide multiple analysis and code transformation optimizations within its repertoire. They also provide a standard set of default optimizations pipelines which that have been generated for a standard benchmark on standard architectures. The ordering of the passes is fixed within the pipelines but the selection of which passes to apply is made by a heuristic cost function. Predictive models can make these decisions by making predictions over which passes to apply or which sub-sequence of passes to apply to a program. Optimization passes can implicitly fed as input to a model by applying them on programs and then training the program representations or explicitly by converting the passes to a bit vector representation [68]. The optimization sequences are represented by bit vectors where an optimization with an on/off switch is represented by a single bit, either 0 or 1. Optimizations with numerical values are represented by its binary representation of all possible values, for example optimization with 3 possible values is represented with (0,0,0),(0,0,1), and (0,1,0) for values 1,2,and 3 respectively.

1) Selection of Optimizations: The selection of an optimization to be applied to a program can be made with a predictive model or a cost model. Multiple such prediction models can construct a sequence of optimizations that can be applied to the said program [53]. A sequence predictor model can be constructed that is trained on program counter value representation. The model performs classification over a subset or phase of optimization passes. Multiple models are used for the different sub-group of passes as shown in Fig.8. A sequence can be generated by combining the predictions

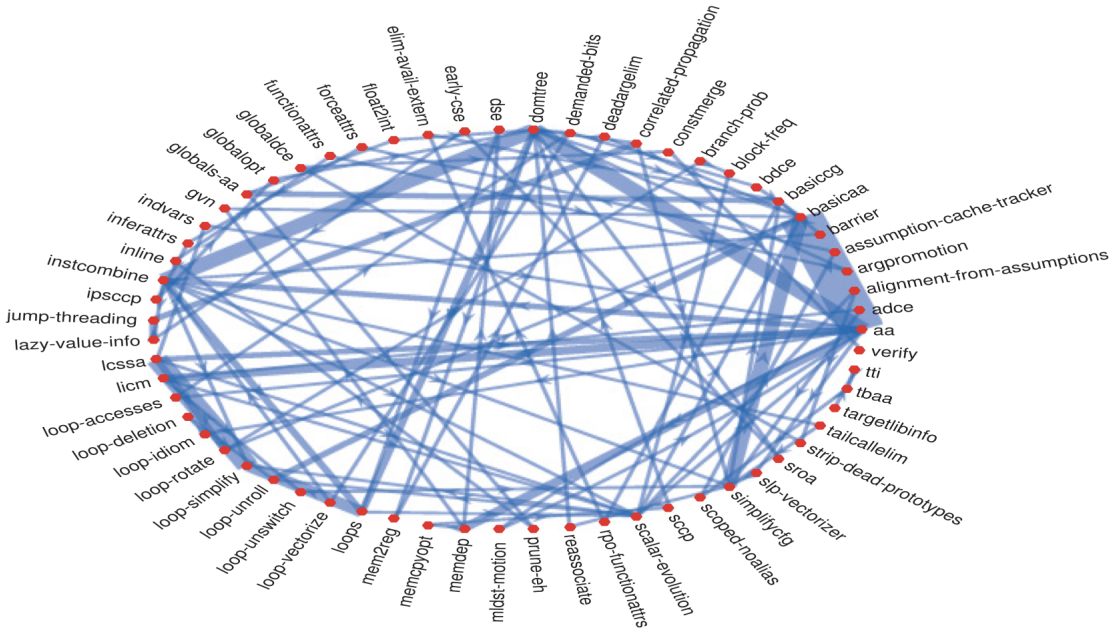


Fig. 7. The dependence graph of LLVM -03's pipeline of optimization. The nodes encode the optimization pass itself and the edges encode their inter-dependencies. [68]

Optimization Phase	List of Optimizations
LNO	blocking-size, cs1, cs2, fission, full-unroll, fusion, interchange, ou-prod-max, pf2, prefetch, prefetch-ahead, simd, trip-count
WOPT	aggcm-threshold, aggstr, value-numbering, combine, dce-aggressive, iv-elimination, spre, canon-expr
OPT	alias, align-padding, div-split, goto, ptr-opt, swp, unroll-size, unroll-times-max
CG	cflow, local-sched-alg, ptr-load-use, use-prefetchnta
GRA	optimize-boundary, prioritize-by-density
TENV	frame-pointer
IPA	callee-limit, ctype, dve, field-reorder, space, plimit,pu-reorder, small-pu, min-hotness

Fig. 8. Search space of optimizations from Open64 compiler. The optimizations are divided into 7 phases shown in the left column. The optimizations in the right column are used to create a sequence prediction model. [53]

TABLE VI
OPTIMIZATION PASSES AVAILABLE WITH THE JIKES JVM JIT COMPILER USED AS SEARCH SPACE OF OPTIMIZATION IN [70]

Optimization Level O0
Local common sub expression elimination
Local constant propagation
Local copy propagation
Control Flow Graph Structural Analysis
Escape Transformations
Field Analysis
Basic block frequency estimation
Optimization Level O1
Branch optimizations
Tail recursion elimination
Basic block static splitting
Simple optimizations like Type prop, Bounds check elimination, dead-code elimination
Optimization Level O2
Loop normalization
Loop unrolling
Coalesce Moves

of multiple models. Multiple sequences of optimizations can be obtained by sampling the probability distribution of the predictions of the model. Though such a predictive model does not encode the impact of a given optimization on the next optimization in a sequence.

To encode the impact of an optimization on a successive optimization, model predictors need to make intermediate predictions for selecting optimizations to apply. A neural network model can be constructed to predict the effect of the successive optimization [41]. Multiple such models are used in an LLVM pipeline in such a way that for each model new static features are extracted to encode the effects of the previous optimization passes. At each model, the results of the previous n optimization passes are combined with the new features to predict if the next n optimization passes should be applied. This generates an optimization selection process in a sequence such that only passes that will optimize the code will be selected.

2) Phase-ordering of Optimizations: The task of phase-ordering of optimizations involves determining an order in which the optimizations are applied. An optimal ordering of optimizations is critical as optimizations can transform pro-

sub-seq	Compiler Passes
A	-ipsccp -globalopt -deadargelim -simplyfycfg -functionattrs -argpromotion -sroa -jump-threading -reassociate -indvars -mldst-motion -lcssa -rpo-functionattrs -bdce -dse -inferattrs -prune-eh -alignment-from-assumptions -barrier -block-freq -loop-unswitch -branch-prob -demanded-bits -float2int -forceattrs -loop-idiom -globals-aa -gvn -loop-accesses -loop-deletion -loop-unroll -loop-vectorize -sccp -strip-dead-prototypes -inline -globaldce -constmerge
B	-licm -mem2reg
C	-loop-rotate -instcombine -loop-simplyfy
D	-memcpyopt
E	-loop-unswitch -adce -slp-vectorize -tailcallelim

Fig. 9. The clustered sub-sequences of optimizations created from LLVM's -03 pipeline in [68]. The combinations of the sub-sequences (Example, BCDE, ABBC) forms the search space of optimizations. [68]

grams that impact the application of a successive optimization. Phase ordering can be modeled as a Markov Process where a heuristic makes a decision on the optimization to perform given the current state of the features of a program [70]. An artificial neural network is modeled as the heuristic function trained on a set of static representations of methods of a program to predict the application of an optimization. One of the outputs of the network consists of a neuron that represents the end of the optimization loop. After applying a predicted optimization, new static features are extracted for the same method, and a new prediction is made using the network. This feedback loop induces an optimal ordering of the optimizations to be applied to the method of a program. The search space of optimizations, also shown in Table VI, consists of the Level 00, 01, and 02 of the Jikes RVM JIT compiler.

In order to focus on the phase-ordering problem of optimizations, the LLVM -03 pipeline can be clustered and each cluster can be represented with a alphabetical character as shown in Fig.9. The combinations of these characters, including repetitions, generates a search space of the ordering of optimization passes. A predictive model can be constructed using program representations extracted after applying the different ordering of passes [68] along with bit vector representation of the passes. The model can predict the execution time speedup as a result of applying a given ordering of passes. The ordering that leads to the prediction of the fastest speedup is considered the optimal ordering.

The ordering of the LLVM -03 pipeline can also be achieved by modeling the tuning problem as a reinforcement learning problem [71]. The agent is modeled as a deep residual neural network that predicts the optimization to apply given an input representation of a program's static intermediate representation features. The actions can be defined with various granularity levels such as sub-sequences of the -03 pipeline, individual passes, and individual passes with parameter values. The state information used by the agent consists of the intermediate representation features and the history of previously applied actions or optimizations. A positive reward is provided for speedup and a negative reward is provided for a slowdown. Through the stages of exploration and exploitation the agent learns to predict an optimal ordering of the optimization passes.

C. Loop optimizations

1) Tuning individual optimizations: Loop unrolling is a loop transformation method that replicates the loop body of a program. The application of this transformation also enables the application of other compiler optimizations. Two factors guard the application of this transformation; if the transformation should be applied and what should be an optimal unroll factor. The unroll factor is a parameter of the transformation that determines by what degree to unroll a given loop. Predictive models can be used to determine if loop unrolling as a transformation should be applied and what unroll factor needs to be used. The application of the transformation can be determined by training a decision tree using static features of loops of a program [20]. The model predicts categories of transformation application potential in terms of improved performance, equal performance, insignificant performance, or degraded performance. Static features can also be used to create model to predict the unroll factor to be used for a given loop [10].

Loop Tiling or cache blocking in another loop transformation method that can be used to improve the performance of loops in a given program. This transformation converts a loop into an outer and an inner loop such that tilling partitions the iteration space while changing the order of execution but not changing the computation. This transformation improves the cache locality of a program such that data reuse is improved and the costly access to lower levels of memory hierarchy is reduced. The optimality of the tile size used is dependent on the data being accessed and the size of the memory hierarchy. A predictive model can be constructed that can effectively estimate the cost in terms of execution time of using a given tile size for a given program on an architecture. Synthetically tiled programs can be generated to train a predictive model where the programs are characterized by the number and type of references in the innermost statements of loops [72]. These features can capture the effects of the spatial and temporal locality of the tiled programs. Predictive models can also be constructed as a regression model that predicts the optimal tile size to be used instead of predicting the estimated execution time of a program. Dynamic representations such as hardware performance counter values can be used to predict tile sizes using an artificial neural network [29].

2) Tuning the search space of loop transformations: Compilers offer many loop specific optimizations that can be applied to the loops of a program. These optimizations are applied by the compiler based on its internal heuristics and the information from the intermediate representation of a program. Predictive models can be built to estimate the cost of applying a loop optimization. Such models need the ability to characterize the loops in a program as feature vectors so that they can be provided as training samples to a predictive model. The previously seen program representations such as the static features, intermediate representation, and program

TABLE VII
SEARCH SPACE OF LOOP OPTIMIZATIONS USED IN HERCULES FRAMEWORK [73]

Optimization	Value Used
Unrolling with factor	0 (no unrolling), 2, 4, 8
Loop fusion distribution	nofuse, maxfuse, smartfuse
Loop tiling	1 (no tiling) or 32
Wavefronting	on/off
Thread-level parallelization	on/off
Pre-vectorization	on/off
SIMD-level parallelization	on/off

```
#P0
#P1
#P2
#pragma clang loop(i,j,k) tile sizes(#P3,#P4,#P5)
    floor_ids(i1,j1,k1) tile_ids(i2,j2,k2)
#pragma clang loop id(i)
  for (i = 0; i < _PB_N; i++) {
    #pragma clang loop id(j)
    for (j = 0; j < _PB_M; j++) {
     #pragma clang loop id(k)
        for (k = 0; k <= i; k++)
        {
          C[i][k] += A[k][j]*alpha*B[i][j]
           + B[k][j]*alpha*A[i][j];
        }
    }
  }
```

Fig. 10. The optimization space of PolyBench's syr2k kernel. The variables P3,P4 and P5 represent the configuration space of the tile sizes for the three loops. [74]

counter values representations may not effectively characterize loops as they have limited loop information such as the data dependency between loops and the memory access patterns in the loop body. HERCULES [73] provides a pattern based search approach to extract loop-specific features that encode characteristics such as memory access patterns of arrays and data dependencies with loops. The loop based features of a program are combined with loop specific optimizations shown in Table VII to predict the speedup of a program on the application of an optimization.

Loop optimizations or transformations can also be applied without creating program representations [74]. The transformations are implemented in [75] as OpenMP-style pragma directives in LLVM/Clang using the LLVM Polly framework. The implemented transformations include loop tiling, interchange, reversal, in addition to Clang provided transformations such as loop unrolling, unroll-and-jam, and vectorization. The pragmas are inserted into the source code of a program on loops that will be a target of optimizations. The search space of transformations is represented using pragma directives with configuration space variables. The variables represent the multiple available configurations for the transformations. An example of the search space is provided in Fig.10. The optimal configuration of the transformations can be found using a Bayesian Optimization tuner such as YTOPT [76]. Bayesian Optimization involves the online learning of a surrogate model which guides the exploration and exploitation search through the optimization configuration space. The surrogate model in use is the Random Forrest Model. The search is evaluated by the execution of the underlying program to find profitable configurations.

D. Code Parallelization

OpenMP is a popular programing model for porting serial implementations of scientific applications to parallel implementations to be used on shared-memory multi and many-core precessors. It lets users access the OpenMP API via pragmas that can be inserted in C, C++, and Fortran programs. It supports parallelization by targeting the compute-expensive loops in a given program. Simply annotating a program with parallelization pragmas can be insufficient to extract maximum performance benefits of the parallelization. The pragmas made available by OpenMP support multiple parameters that can be tuned to achieve performance benefits. The available parameters give rise to the selection of their multiple configurations which can vary with the input size of the program. Manually searching through the configurations might be easier for cases when the search space is small but still requires multiple execution which may be costly in the case of larger scientific applications. Predictive models using machine learning or deep learning can alleviate the search and the cost of the search of configurations by acting as cost models to make predictions over the configurations of the pragma parameters.

1) Parallelization Detection: The first step in introducing parallelization is the identification of regions in a code that can benefit from parallelization. Programming models such as OpenMP provide user-insertable pragmas in code but rely on the user's expertise and time-consumption to identify and introduce parallelism efficiently. Modern compilers can detect parallelism from static (Pluto [25], AutoPar [77]), dynamic, and hybrid(DiscoPop [78]) analysis methods but these methods can be conservative in the case of static analysis and may require program executions in the case of dynamic analysis. Predictive modeling methods offer the opportunity to construct a cost function that can predict the parallelization effectiveness of a given program region.

The effectiveness of parallelization for a program can be modeled by constructing a classification model [79]. The classification model predicts the binary decision of the application of the parallelization directive(#pragma omp parallel for). The programs from the NAS Parallel Benchmark [14] are pre-annotated with OpenMP directives and can be represented by dynamic features extracted using DiscoPop [78]. The features are used to train an SVM model or a decision tree based model if the classification rules need to be learned.

Loops exhibit patterns based on memory access dependencies in their iterations which can be used to detect loops that can be parallelized [80]. Loops have patterns such as loop iterations that do or do not have memory access dependencies between their iterations and stencil patterns where a neighborhood of elements are updated by a loop iteration access. These patterns are detected by modeling a relational graph convolutional neural network with a graph based representa-

TABLE VIII
THE SEARCH SPACE OF OPENMP DIRECTIVES AND THEIR ASSOCIATED PARAMETERS USED IN [80]

Search Space (Skylake Processor)	**Parameter Values**
Number of threads	1, 2, 3, 4, 5, 6, 7, 8
Scheduling Policy	STATIC, DYNAMIC, GUIDED
Chunk Sizes	1, 8, 32, 64, 128, 256, 512
Search Space (SandyBridge Processor)	**Parameter Values**
Number of threads	1, 2, 3, ... , 32
Thread Affinity	close, spread

tion, hardware performance counters, and sample input sizes of a program. The graph convolutional network representation is further used by a fully connected layer that uses the output of the graph network, the performance counters, and the input size of a program to predict the pattern type of a loop.

2) Tuning Parallelization Parameters: Once parallelizable regions are detected in a program, pragma directives can be added to the region of the program. The pragma directives provide parameters that can be selected to tune the parallelism of the said region. As such, it provides a tuning opportunity for predictive models to make decisions that can improve parallelization on a given architecture.

Predictive models can be used to create a cost function that can determine if loop candidates are parallelizable and the kind of scheduling policy that needs to be used [81]. Not all loops provide performance benefits upon parallelization and the scheduling policy that assigns tasks to threads can be inefficient. This gives rise to a tunable opportunity exploited by [81] that trains an SVM with a hybrid representation of static and dynamic representations of a program to predict parallelizable loops and an optimal scheduling policy.

Parallel regions in a code like loops exhibit patterns based on their memory dependencies between iterations. These patterns are used to detect parallelizable loops and then used to tune the pragma-directives applied to the said loops [80]. A graph convolutional neural network is used with a fully connected layer to predict the number of threads to be used, the optimal scheduling policy type, the associated chunk size to be used, or the thread affinity options to distribute work to the individual cores. An optimal thread size leads to maximizing the parallelizability of loop iterations whereas tuning the other parameters helps with load distribution between the threads. A numerical or categorical configuration search space is built for the parameters based on the OpenMP implementation and the underlying architecture. Graph2Par [82] uses its graph transformer based model built by representing loops in programs with the abstract syntax tree based representation infused with information from control flow graphs to make predictions on the pragma directives to be used. These directives include parallel for, parallel reduce, SIMD-vectorization enabling and target offloading.

Parallel programs may not run in isolation and may have to share resources with other applications. In such scenarios the external workload needs to be characterized to predict an optimal mapping of a program to threads. A predictive model can be built as a cost function to determine the optimal amount of threads to be used for a parallel region instead of simply assigning the number of threads equal to the number of available cores [83] [84]. The number of threads to be used can be decided at runtime by combining the static representation of a program and state of the runtime environment characterized by the number of processes waiting for scheduling and average load on the system.

E. Heterogenous Computing Offloading

The programming model OpenMP version 4 introduces the ability to offload computations to a target device such as GPUs and FPGAs. It offers a portable method using pragma directives injected into the source code to exploit these accelerator devices. This provides end-users to run optimal programs on a diverse set of architectures that support different kinds of accelerator devices. OpenMP provides a clause called target for offloading code sections to highly-parallel accelerators like GPUs. The offloading consists of mapping data between the host and the device and then offloading computations from the host to the device.

To maximize the performance on the highly-parallel GPUs, OpenMP provides additional clauses to define how the parallel tasks are spread across threads. Threads are grouped into a team using the teams clause and groups of teams are scheduled using the distribute clause. Multiple variants of a loop in a program can be created to define the distribution of tasks within a team and across groups. Additional variants can be created by using loop interchange and loop collapse transformations. These parallel loop transformation variants form a search space that can be tuned for a given loop. A speedup predictor [85] is built by characterizing loops in a program with static features and then training the predictor with the different transformations applied to the program. This builds a cost model to make predictions on the cost of offloading programs to GPUs and the transformations that maximize performance.

The Rigel Framework [86] as part of its OpenMP performance tuning infrastructure includes the modeling of OpenMP target offloading pragma directive. It builds an ensemble model with decision trees to make binary predictions on the cost benefit of offloading loops to a GPU. The cost benefit is defined by the ratio of GPU throughput and CPU throughput greater than 1.1x, where the GPU throughput characterizes the offloading overheads which include memcpy from the host to device and vice-versa. The decision tree based model is built by characterizing programs based on their GPU-offloading specific architecture-independent representations. The decision tree based predictions are followed by a heuristic based tuning of the individual parameters of thread_limit, number of teams of threads, and collapse pragma directive options.

F. Code Similarity

The process of tuning code similarity involves the selection of alternative programs or code variants that can improve

performance on a given architecture. The alternative programs could contain variations such as different representation of matrices in code kernels, a different implementation of an algorithm such as breadth-first search, or pre-optimized loop computations in a code kernel. A constraint defined code variants is that they should be functionally invariant to the initial code but achieve better performance in metrics such as execution time. The variants can be explicitly tuned by expert programmers and be made available as a part of libraries such as Breadth-First Search (BFS) implementations from Back40 library [87], CUDA's Histogram building implementations in Unbound [88] library, Sparse Matrix-Vector Multiplication (SPMV) variants in CUDA's CUSP [89] libraries for GPUs. Variants can also be generated by iterative compilation using search heuristics with frameworks such as Orio. The process of matching code's with performant variants involves building a code matching model which can be achieved with machine learning and deep learning models.

A code similarity matching model can be implemented using machine learning models such as an SVM. The SVM is trained to learn a function that maps the representations of a code to that with its performant code variants that improve its execution time [90]. The code representations are characterized by the input data set features for a given specific code and they are mapped to labels that span the distribution of the available code variants. For example, the SPMV code kernel is represented by features related to its matrix row length such as average non-zeros per matrix row, whereas BFS implementations are represented with graph based features consisting of number of nodes and edges, and the average out-degree of a node. The similarity matching model can be further enhanced by incorporating information from the underlying architecture [91]. This enables the portability of the code similarity model to newer architectures. Various GPU architectures can be represented with features based on the architecture itself. These features are extracted from CUDA's deviceQuery program that represent statistics such as the number of cuda cores, L2 cache size, and shared memory per block. The Nitro Framework is enhanced by concatenating architecture-based features with the input data set features to construct a multi-task learning model where each task represents a source training architecture. This modification of the Nitro framework enables learning code similarity on multiple architectures and then making predictions on newer unseen architecture thereby introducing portability.

The Nitro Framework proposed the use of features that are related to the input data of a code variant to be tuned. Though it requires expert programmers to develop the set of features that are representative of a code kernel. The Meliora framework [22] proposes an automated method to build a dataset of code kernel representations and their associated variants. The code kernel variants consist of optimized loops using loop transformations in a program which are generated using the Orio autotuner. The framework uses a Graph Convolutional Neural Network (GCNN) [92] to learn a mapping between unoptimized code kernel and its associated optimized variants. The code kernels are represented by a Control-Flow Graph enhanced with instruction counts for its nodes and execution probabilities on its edges. The optimized variant predicted by the GCNN can be used as is without further modifications or the optimizations in the matched code kernel can be used as a starting point for further optimizations using an autotuner. The matching model enables the reduction of the search space of loop optimizations.

Both the Nitro and Meliora framework models match a code kernel with a variant code kernel but the models are limited to making predictions on a pre-decided distribution of variant code kernels. Performance embeddings [23] is a method to represent a code kernel such as loop nests in the form of embeddings. The embeddings are generated by combining its static features, performance counter values, and intermediate representation based graphs. The features are combined using a combination of neural network and a graph transformer. The embedding representation allows their storage for downstream tasks. The embeddings can be stored in a look up table with their associated optimizations. A look up table allows the matching of embeddings using similarity metrics such as cosine similarity with embeddings of unknown code kernels. Furthermore, K-means clustering can be used to reduce the search space to a clustered neighborhood. Embeddings of unknown code kernels can be matched with embeddings of known kernels to potentially utilize their associated loop optimizations.

V. Conclusion

This survey discusses the need for ML/DL-based predictive models to replace the cost functions of hand-crafted heuristic models in compiler tuning. It further discusses the utility of using the models in making direct predictions on program optimizations. The models can be used to automate the process (autotuning) of tuning said optimizations. Training these models requires the collection of benchmark datasets, feature engineering, and utilizing a metric to gauge the model performance. Feature engineering is used to construct representative features to characterize programs or code kernels that would be the subject of optimizations. As various sources of features are available, their categorization is required, as done in this survey. The choices of available optimizations can be extensive based on the area where predictive models are employed, and therefore, their search space needs to be defined. We then discuss the various areas of autotuning and their associated search space.

The area of automating program optimization will continue to grow as software applications adapt to the evolution of hardware. While this gap exists, it gives autotuner developers ample opportunities to extract maximum performance from the underlying hardware architecture. The progression of autotuning software from empirical search models to machine learning models has reduced the overheads from repeated program execution. The progression from machine learning to deep learning models has alleviated some of the burden of engineering program representative features. The datasets used

to train models have consisted of benchmarks representative of common subroutines in larger applications such as matrix-matrix product in deep learning computations. While these subroutines can approximate unseen programs to a degree, they may not represent the larger distribution of programs written by programmers. Synthetic datasets such as CLgen [42] and Genesis [93] have been proposed that could be beneficial for data-hungry deep learning models.

The areas of program optimization in compiler passes, loop optimizations, and heterogeneous compute offloading have seen significant research efforts. The space complexity and the complex interactions of optimizations will continue to grow as new compiler optimization passes or OpenMP directives are added to the search space. The introduction of new programming models and their associated hardware, if any, will continue to provide opportunities for optimizations. Performance Portable frameworks such as RAJA [94] and KOKKOS [95] [96] provide new tuning opportunities as they provide methods to write a single program or application for execution on multiple programming systems defined as execution policies. Apollo [97] and Artemis [98] are two such autotuners that construct machine learning models in an offline-trained and online-trained setting to make predictions on selecting an optimal execution policy. The Kokkos framework provides other unexplored tuning opportunities, such as its data mapping policies and parameters for implementing hierarchical parallelism.

The rapid development of machine learning and deep learning models over the years has propelled the area of autotuning using predictive models. Graph-based representations and hardware counter tools such as PAPI are currently the preferred choice for representing programs. The popularity of the former could be attributed to their ability to encode data-flow and control-flow dependencies and the abilities of deep learning models to exploit graph-based structures. The graph representation will continue to be a playground for experimentation as its structure can encapsulate various information about a program. The development of node-based instruction embeddings [48] and the use of newer deep learning models such as transformers [23] will lead the way for such experimentation. The popularity of hardware counters is attributed to extracting representations that model the runtime behavior of programs with different input data sizes. The polyhedral model has been used for applying loop optimizations, but it has the unexplored potential to represent programs using its schedule statement and memory access patterns. Tools such as MICA and AIWC have been introduced to provide architecture-independent program representations, but their portability has been demonstrated in limited works.

A. Future with Large Language Models

The advent of Large Language Models (LLM) will reintroduce the question of whether they can be used for optimizing programs by directly using the source code of a program. While LLMs have been used for code-related tasks such as code completion with Codex [99] in Copilot [100] they have not been used for code-performance optimization. The task of code optimization has been recently experimented with the Llama 2 [101] LLM wherein LLVM IR is used as the source and target language instead of the source code of a program [102]. The LLM is trained on unoptimized LLVM IR as prompts to produce optimization passes and optimized IR as answers. The objective of the LLM is to reduce the program's code size by producing an ordering of optimization passes or directly producing the optimized IR. The length of token sequences currently limits the model and hence limits the training to smaller code fragments. The usability of LLMs with just the source code as input and its portability across hardware architectures remain future research questions to explore. LLMs could be ideal for programmer productivity as their user experience can be far simpler than the existing solutions.

REFERENCES

[1] B. J. Gough and R. Stallman, *An Introduction to GCC.* Network Theory Limited, 2004.

[2] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[3] L. S. Basics, "Intel® math kernel library," 2005.

[4] I. Advanced Micro Devices, "Amd optimizing cpu libraries." [Online]. Available: https://www.amd.com/en/developer/aocl.htmldocumentation

[5] N. Corporation, "cublas: Gpu-accelerated implementation of the basic linear algebra subroutines (blas)." [Online]. Available: https://developer.nvidia.com/cublas

[6] ——, "cudnn: Gpu-accelerated library of primitives for deep neural networks (dnn)." [Online]. Available: https://developer.nvidia.com/cudnn

[7] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[8] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology," in *ACM International Conference on Supercomputing 25th Anniversary Volume*, 1997, pp. 253–260.

[9] M. Frigo and S. Johnson, "Fftw: an adaptive software architecture for the fft," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 3, 1998, pp. 1381–1384 vol.3.

[10] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *International symposium on code generation and optimization*. IEEE, 2005, pp. 123–134.

[11] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[12] P. Balaprakash, S. M. Wild, and B. Norris, "Spapt: Search problems in automatic performance tuning," *Procedia Computer Science*, vol. 9, pp. 1959–1968, 2012.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[14] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon, "Nas parallel benchmark results," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 43–51, 1993.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[17] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in *2009 ieee international symposium on parallel & distributed processing*. IEEE, 2009, pp. 1–11.

[18] F. Li, F. Tang, and Y. Shen, "Feature mining for machine learning based compilation optimization," in *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE, 2014, pp. 207–214.

[19] L. Benini, A. Bogliolo, M. Favalli, and G. De Micheli, "Regression models for behavioral power estimation," *Integrated Computer-Aided Engineering*, vol. 5, no. 2, pp. 95–106, 1998.

[20] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *Artificial Intelligence: Methodology, Systems, and Applications: 10th International Conference, AIMSA 2002 Varna, Bulgaria, September 4–6, 2002 Proceedings 10*. Springer, 2002, pp. 41–50.

[21] A. Brauckmann, A. Goens, S. Ertel, and J. Castrillon, "Compiler-based graph representations for deep learning models of code," in *Proceedings of the 29th International Conference on Compiler Construction*, 2020, pp. 201–211.

[22] K. Meng and B. Norris, "Guiding code optimizations with deep learning-based code matching," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2020, pp. 20–28.

[23] L. Trümper, T. Ben-Nun, P. Schaad, A. Calotoiu, and T. Hoefler, "Performance embeddings: A similarity-based transfer tuning approach to performance optimization," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 50–62.

[24] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc, "Autotuning in high-performance computing applications," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2068–2083, 2018.

[25] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[26] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[27] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[28] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. Benatchba *et al.*, "A deep learning based cost model for automatic code optimization," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 181–193, 2021.

[29] A. M. Malik, "Optimal tile size selection problem using machine learning," in *2012 11th International Conference on Machine Learning and Applications*, vol. 2. IEEE, 2012, pp. 275–280.

[30] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan, "Adapt: A framework for coscheduling multithreaded programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–24, 2013.

[31] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for gpu program optimizations," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–10.

[32] 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*, 2014, pp. 455–466.

[33] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. De Supinski, and M. Schulz, "Identifying energy-efficient concurrency levels using machine learning," in *2007 IEEE International Conference on Cluster Computing*. IEEE, 2007, pp. 488–495.

[34] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 661–672.

[35] A. Dutta, J. Choi, and A. Jannesari, "Power constrained autotuning using graph neural networks," *arXiv preprint arXiv:2302.11467*, 2023.

[36] A. F. Zanella, A. F. da Silva, and F. M. Quintão, "Yacos: a complete infrastructure to the design and exploration of code optimization sequences," in *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*, 2020, pp. 56–63.

[37] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[38] R. Mosaner, D. Leopoldseder, L. Stadler, and H. Mössenböck, "Using machine learning to predict the code size impact of duplication heuristics in a dynamic compiler," in *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2021, pp. 127–135.

[39] 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*, vol. 39, pp. 296–327, 2011.

[40] J. Cavazos and M. F. O'boyle, "Method-specific dynamic compilation using logistic regression," *ACM SIGPLAN Notices*, vol. 41, no. 10, pp. 229–240, 2006.

[41] T. Jayatilaka, H. Ueno, G. Georgakoudis, E. Park, and J. Doerfert, "Towards compile-time-reducing compiler optimization selection via machine learning," in *50th International Conference on Parallel Processing Workshop*, 2021, pp. 1–6.

[42] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "Synthesizing benchmarks for predictive modeling," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 86–99.

[43] ——, "End-to-end deep learning of optimization heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2017, pp. 219–232.

[44] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[45] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "Neurovectorizer: End-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 242–255.

[46] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[47] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 196–206.

[48] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[49] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 2007, pp. 185–197.

[50] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[51] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Proc. of the 2006 Ottawa Linux Symposium*. Citeseer, 2006, pp. 269–288.

[52] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[53] E. Park, S. Kulkarni, and J. Cavazos, "An evaluation of different modeling techniques for iterative compilation," in *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, 2011, pp. 65–74.

[54] 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*, 2009, pp. 75–84.

[55] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[56] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE micro*, vol. 27, no. 3, pp. 63–72, 2007.

[57] Y. Li and T. Li, "Bioinfomark: A bioinformatic benchmark suite for computer architecture research," *ECE, University of Florida, Tech. Rep*, 2005.

[58] A. H. Ashouri, A. Bignoli, G. Palermo, and C. Silvano, "Predictive modeling methodology for compiler phase-ordering," in *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, 2016, pp. 7–12.

[59] A. H. Ashouri, G. Mariani, G. Palermo, and C. Silvano, "A bayesian network approach for compiler auto-tuning for embedded processors," in *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. IEEE, 2014, pp. 90–97.

[60] B. Johnston and J. Milthorpe, "Aiwc: Opencl-based architecture-independent workload characterization," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2018, pp. 81–91.

[61] B. Johnston, G. Falzon, and J. Milthorpe, "Opencl performance prediction using architecture-independent features," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 561–569.

[62] K. Vaswani, M. J. Thazhuthaveetil, Y. Srikant, and P. Joseph, "Microarchitecture sensitive empirical models for compiler optimizations," in *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 2007, pp. 131–143.

[63] E. Park, J. Cavazos, L.-N. Pouchet, C. Bastoul, A. Cohen, and P. Sadayappan, "Predictive modeling in a polyhedral optimization space," *International journal of parallel programming*, vol. 41, no. 5, pp. 704–750, 2013.

[64] L. G. Martins, R. Nobre, J. M. Cardoso, A. C. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–28, 2016.

[65] L. G. Martins, R. Nobre, A. C. Delbem, E. Marques, and J. M. Cardoso, "Exploration of compiler optimization sequences using clustering-based selection," in *Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, 2014, pp. 63–72.

[66] R. Cilibrasi and P. M. Vitányi, "Clustering by compression," *IEEE Transactions on Information theory*, vol. 51, no. 4, pp. 1523–1545, 2005.

[67] J. Felsenstein, "Inferring phylogenies," in *Inferring phylogenies*, 2004, pp. 664–664.

[68] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–28, 2017.

[69] S. Cereda, G. Palermo, P. Cremonesi, and S. Doni, "A collaborative filtering approach for the automatic tuning of compiler optimisations," in *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2020, pp. 15–25.

[70] S. Kulkarni and J. Cavazos, "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*, 2012, pp. 147–162.

[71] R. Mammadli, A. Jannesari, and F. Wolf, "Static neural compiler optimization via deep reinforcement learning," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2020, pp. 1–11.

[72] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 190–199.

[73] E. Park, C. Kartsaklis, and J. Cavazos, "Hercules: Strong patterns towards more intelligent predictive modeling," in *2014 43rd international conference on parallel processing*. IEEE, 2014, pp. 172–181.

[74] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, "Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, p. e6683, 2022.

[75] M. Kruse and H. Finkel, "User-directed loop-transformations in clang," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2018, pp. 49–58.

[76] X. Wu, P. Balaprakash, M. Kruse, J. Koo, B. Videau, P. Hovland, V. Taylor, B. Geltz, S. Jana, and M. Hall, "ytopt: Autotuning scientific applications for energy efficiency at large scales," *arXiv preprint arXiv:2303.16245*, 2023.

[77] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.

[78] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf, "Discopop: A profiling tool to identify parallelization opportunities," in *Tools for High Performance Computing 2014: Proceedings of the 8th International Workshop on Parallel Tools for High Performance Computing, October 2014, HLRS, Stuttgart, Germany*. Springer, 2015, pp. 37–54.

[79] D. Fried, Z. Li, A. Jannesari, and F. Wolf, "Predicting parallelization of sequential programs using supervised learning," in *2013 12th international conference on machine learning and applications*, vol. 2. IEEE, 2013, pp. 72–77.

[80] A. Dutta, J. Alcaraz, A. TehraniJamsaz, A. Sikora, E. Cesar, and A. Jannesari, "Pattern-based autotuning of openmp loops using graph neural networks," in *2022 IEEE/ACM International Workshop on Artificial Intelligence and Machine Learning for Scientific Applications (AI4S)*. IEEE, 2022, pp. 26–31.

[81] Z. Wang, G. Tournavitis, B. Franke, and M. F. O'boyle, "Integrating profile-driven parallelism detection and machine-learning-based mapping," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–26, 2014.

[82] L. Chen, Q. I. Mahmud, H. Phan, N. Ahmed, and A. Jannesari, "Learning to parallelize with openmp by augmented heterogeneous ast representation," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[83] M. K. Emani, Z. Wang, and M. F. O'Boyle, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[84] D. Grewe, Z. Wang, and M. F. O'Boyle, "A workload-aware mapping approach for data-parallel programs," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011, pp. 117–126.

[85] A. Mishra, S. Chheda, C. Soto, A. M. Malik, M. Lin, and B. Chapman, "Compoff: A compiler cost model using machine learning to predict the cost of openmp offloading," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 391–400.

[86] P. Rameshka, P. Senanayake, T. Kannangara, P. Seneviratne, S. Jayasena, T. Rusira, and M. Hall, "Rigel: A framework for openmp performancetuning," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2019, pp. 2093–2102.

[87] N. Corporation, "Cuda unbound." [Online]. Available: https://nvlabs.github.io/cub/

[88] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *Computer Vision–ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part I 10*. Springer, 2008, pp. 304–317.

[89] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[90] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 501–512.

[91] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai, "Architecture-adaptive code variant tuning," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 325–338, 2016.

[92] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International conference on machine learning*. PMLR, 2016, pp. 2014–2023.

[93] A. Chiu, J. Garvey, and T. S. Abdelrahman, "Genesis: a language for generating synthetic training programs for machine learning," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015, pp. 1–8.

[94] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 2019, pp. 71–81.

[95] H. C. Edwards, "Enabling performance portability across manycore architectures (kokkos)." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2014.

[96] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[97] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Reusable models for fast, dynamic tuning of input-dependent code," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 307–316.

[98] C. Wood, G. Georgakoudis, D. Beckingsale, D. Poliakoff, A. Gimenez, K. Huck, A. Malony, and T. Gamblin, "Artemis: Automatic runtime tuning of parallel execution parameters using machine learning," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings 36*. Springer, 2021, pp. 453–472.

[99] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[100] Microsoft, "Copilot." [Online]. Available: https://copilot.github.com/

[101] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[102] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve *et al.*, "Large language models for compiler optimization," *arXiv preprint arXiv:2309.07062*, 2023.