A Hybrid Machine Learning Model for Code Optimization

Yacine HAKIMI[1*], Riyadh BAGHDADI[2] and Yacine CHALLAL[3]

[1*] Laboratoire de Méthodes de Conception des Systèmes(LMCS), Ecole nationale Supérieure d'Informatique, Oued Smar, BP M68,16309, Algeria.
[2]New York University Abu Dhabi (NYUAD), Abu Dhabi, UAE.
[4] College of Computing and Information Technology, University of Doha for Science and Technology (UDST), Doha, Qatar.

*Corresponding author(s). E-mail(s): y_hakimi@esi.dz;
Contributing authors: baghdadi@nyu.edu;
yacine.challal@udst.edu.qa;

Abstract

The complexity of programming modern heterogeneous systems raises huge challenges. Over the past two decades, researchers have aimed to alleviate these difficulties by employing classical Machine Learning and Deep Learning techniques within compilers to optimize code automatically. This work presents a novel approach to optimize code using at the same time Classical Machine Learning and Deep Learning techniques by maximizing their benefits while mitigating their drawbacks. Our proposed model extracts features from the code using Deep Learning and then applies Classical Machine Learning to map these features to specific outputs for various tasks. The effectiveness of our model is evaluated on three downstream tasks: device mapping, optimal thread coarsening, and algorithm classification. Our experimental results demonstrate that our model outperforms previous models in device mapping with an average accuracy of 91.60% on two datasets and optimal thread coarsening task where we are the first to achieve a positive speedup on all four platforms while achieving a comparable result of 91.48% in the algorithm classification task. Notably, our

approach yields better results even with a small dataset without requiring a pre-training phase or a complex code representation, offering the advantage of reducing training time and data volume requirements.

Keywords: Machine Learning, Deep Learning, Code Optimizations, LLVM-IR Heterogeneous Platforms, Thread Coarsening, Algorithm Classification, Code mapping

1 Introduction

Using heterogeneous systems to perform large-scale computations is becoming increasingly common in academia and industry [1], [2], [3]. However, writing parallel code becomes more challenging when we target such heterogeneous systems, which contain different hardware types of accelerators (CPUs, GPUs, etc.) with diverse performance characteristics and behaviors.

Multiple parameters affect the performance of parallel code in a heterogeneous context, ranging from selecting the algorithm and programming model to determining the granularity of parallelism, the communication pattern, scheduling, and mapping strategy. It is always necessary to find the optimal or near-optimal parameters to achieve better performance gains on a given architecture for a given problem. Some parameters may vary from one system to another or depend on other factors, such as input data. Aiming to hide the complexity of tuning these parameters and simplify parallel code writing, researchers have explored compiler-based approaches that automatically optimize code for a specific architecture [4]. These compilers make optimization decisions, such as selecting the coarsening factor for a given kernel on a GPU [4], [5], [6], [7], or where to map a piece of code (Kernel) to the CPU or GPU[8], [9] [5].

In recent years, the success of Machine Learning, particularly Deep Learning, has led to its increased utilization in compiler and code optimization problems [4], [9], [10]. Deep Learning has demonstrated its ability to improve accuracy and solve feature extraction and selection problems [4], but it requires large amounts of labeled data which is scarce in the compiler and code optimization field. While simple models, like Decision Trees and SVM, require less labeled data, their performance heavily relies on the quality of the features. Generally, a limited number of high-quality features is sufficient for these models to give good results, but extracting these high-quality of features is a nontrivial task and requires expert developers and domain experts[11].

Natural Language Processing (NLP) approaches have greatly influenced automated feature extraction for code using Deep Learning [12] due to similarities between code and natural languages. These approaches, especially pre-trained models such as CodeBERT[13], have become a feasible way for developers to extract code features and reduce the amount of engineering previously needed. They gave state-of-the-art results in code tasks similar to NLP

tasks (code completion, code-to-code translation, code summarization, etc.) [13]. These techniques also have been used in compiler and code optimization [4] [5]. However, compiler and code optimization tasks are different from natural language processing tasks due to the structural nature of the code and the complexity of interactions between programs and architectures, and the fact that features extracted have to characterize the semantics of the program and its behavior on the target architecture.

We propose, in this work, a novel approach that combines traditional Machine Learning and Deep Learning to address the challenge of code optimization. Our method relies on the CNN model's efficiency in extracting a small vector of high-quality features and the effectiveness of traditional Machine Learning with small datasets. By leveraging Deep Learning, we extract code features without feature engineering and selection. Then we use these features as input to classical Machine Learning to search for the best-performed algorithms for the required task.

Our study demonstrates, that even with small datasets, using Deep Learning and traditional Machine Learning together yields results similar to or better than state-of-the-art pre-trained models and complex representations that require a large dataset. Notably, our approach offers the advantage of reduced training time and data volume requirements without a pre-training phase and the need for a complex code representation.

The contributions of this paper can be summarized as follows:

- We propose a new model for code optimization tasks. This model is a hybrid model that uses Deep Learning and classical machine learning.
- We evaluate this model on three downstream tasks: device mapping, thread coarsening, and algorithm classification.
- The proposed model is the first to match and outperform the performance of state-of-the-art pre-trained models without requiring large amounts of data.

2 Background

The target research field of this paper focuses on optimizing code to get better performance on heterogeneous parallel computing systems. For optimization, the compiler or developer has to make many decisions, such as device mapping and optimal thread coarsening. Algorithm classification can help characterize the code to facilitate decisions. The terminology used in this paper will be explained in detail in this section to ensure a comprehensive understanding of the subsequent sections.

2.1 Heterogeneous parallel systems

In general, parallel computing systems can be divided into two main categories based on processor types and other resources used. Systems that use multiple or many identical processor types are called homogeneous, while systems that

use multiple or many different processor types are called heterogeneous. Nowadays, parallel computing systems are dominated by heterogeneous systems composed of a variety of processors with different sizes, speeds, and memory types [14]. The most common systems consist of CPUs combined with specialized processing units or accelerators such as graphics processing units (GPUs) and tensor processing units (TPUs), field programmable gate arrays (FPGAs), etc.

2.2 Code and Compiler optimization

Parallel computing systems offer potentially high performance while increasing the difficulty of writing and optimizing programs [15]. Parallel computing systems offer potentially high performance while increasing the difficulty and complexity of writing and optimizing programs. Because of the variety of processing units in terms of architectures and characteristics, the problem becomes more complex in heterogeneous parallel systems. To attain the desired performance, developers must optimize their code every time the architecture changes, making manual software porting and optimization for diverse parallel architectures impractical.

Since manually optimizing code for diverse parallel architectures is very expensive, researchers are trying to use compilers for automatic code optimization, and this research problem can be divided into two main categories [9]:

- **Optimization selection problem:** In this category, the objective is to decide whether to apply an optimization or not and their parameters for a given parallel architecture. The research problems addressed in this paper fall into this category. An example of such an optimization is thread coarsening, where we must decide whether or not to apply it and select the optimal "coarsening factor" parameter.
- **The phase-ordering problem:** The goal is to determine the ideal ordering of optimizations and their parameters, considering their interactions. This type of problem is not within the scope of this paper.

2.3 Device mapping problem

The goal here is to enable the compiler to decide which processing unit a given piece of code should run on to maximize performance.

2.4 Algorithm classification problem

Program classification by algorithms is beneficial for software engineering tasks [16]. In the context of code parallelization and optimization, this classification can identify what kind of parallel pattern the code does have, and thus the thread communication and the data-sharing behaviors [17]. Such details can help with optimization decisions.

2.5 Thread coarsening problem

Thread coarsening is an optimization where the compiler combines the code instructions of several threads into one. The coarsening factor is the value that determines how many threads to merge together. This code transformation can improve the performance of a particular parallel program in a specific architecture but can yield a slowdown in another one. The goal here is to decide whether to apply this optimization depends on the nature of the code optimized and on the target hardware architecture.

3 Related work

Using machine learning techniques in compiler and code optimization has yielded impressive outcomes, facilitating the prediction of optimal outputs for various optimization problems [11]. Figure 1 outlines a comprehensive overview of how to use machine learning in code optimization. The dataset is generated by compiling and executing each code with every legal parameter of the target optimization problem and getting the performance measurements (see Figure 1 (A)). After selecting the desired code representation Figure 1 (B), a machine learning model is trained using the resulting dataset, as shown in Figure 1 (C). In the first approach, the model is trained to predict the best optimization directly, using the code representation as input. In the second approach, the code representation and the best optimization are the input, and the output is the performance measurement (the execution time). The trained model can then be used to make predictions for new code, Figure 1 (D).

For example, in the context of the thread coarsening problem, the approach is to compile and execute each kernel using every legal coarsening factor. The aim here is to measure the execution time for each combination. Subsequently,

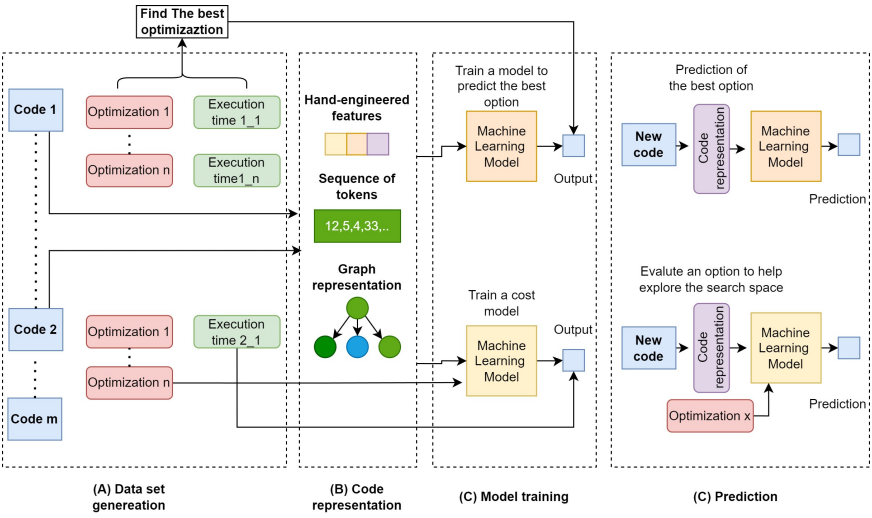

Fig. 1 An overview of the use of machine learning in code optimization

two potential options arise: first, developing a model that takes the kernel as input and directly predicts which coarsening factor yields optimal performance. Or second, creating a model that considers both the kernel with its associated coarsening factor as input and outputs the corresponding execution time. This latter approach effectively utilizes machine learning techniques to explore the search space in order to determine the optimal coarsening factor.

Generally, there are these two approaches to using machine learning in code and compiler optimization [11]:

- Using machine learning as a cost function to evaluate a potential solution provided by other search methods: Many previous work use machine learning to build a cost model to estimate speedup and use another optimization method to search for the optimal solution. For example, in [18], the authors use Deep Learning to predict the speedup of a program after applying a sequence of code transformations and use this model to explore the search space.
- Using machine learning to predict the optimal solution directly: For example, in [19], the authors use a machine learning model to predict the optimal number of threads and the scheduling policy for an OpenMP program.

One of the most important questions when building machine learning systems in compiler optimization is how the code should be represented for the model. In other words, which features will be able to characterize the program to make accurate predictions about the target task? Most recent work addresses this question aiming to improve results by improving representation. Therefore, we categorize the literature depending on how previous research attempted to answer this question.

3.1 Hand-engineered features

Traditional machine learning algorithms take as input a set of features manually extracted from the program. The earlier work in this field relies on classical machine learning algorithms in their work targeting various compiler and code optimizations such as selecting the most beneficial loop unroll factor using SVM and NN (nearest-neighbor) [20], scheduling kernels on CPU/GPU heterogeneous platforms using SVM [21], decision trees [8], determining the optimal work distribution between the CPU and GPU using linear regression [22], determining the best partitioning strategy of irregular applications using k-Nearest Neighbors (KNN) [23] and streaming applications using NN [24], selecting the best value for the coarsening factor on GPU using ANN [6], determining whether parallelism is beneficial using SVM [19], and many other problems. A comprehensive survey can be found in [10].

The performance of these traditional machine learning techniques is greatly affected by the quality of the features. Extracting high-quality of features requires expert intervention and a significant amount of time, and these features are not necessarily efficient in solving other optimization problems, even for the same code. [11].

3.2 Sequence of tokens

Due to the similarities between source code and natural languages and the tremendous success of Deep Learning in NLP, using NLP techniques in compilers and code optimization to solve the feature extraction problem has received significant attention in recent years.

The first work in this direction was by the authors of [4], where they tried to automatically extract features from OpenCL source code using an End-to-End Deep Learning model with embedding techniques to learn a distributed representation of the code. Their model, DeepTune, obtained better results than manual feature extraction in traditional machine learning methods in device mapping and thread coarsening problems.

In [25], the authors have used Deep Reinforcement Learning (DRL) in a system called NeuroVectorizer to predict the optimal vectorization parameters for a given loop in C and C++. They used code2vec to generate the embedding representation by code2vec [26], which initially decomposes the code into a set of pathways in the AST (Abstract syntax tree) to learn the representation of each of them and finally aggregate them.

Inst2vec [5] used the skip-gram model to train an embedding layer analyzing the ConteXtual Flow Graph (XFG) to add more information to the distributed representation. This pre-trained model has generally shown exciting results on device mapping, thread coarsening, and algorithm classification tasks.

Using LLVM IR, the Intermediate Representation (IR) of the LLVM compiler (Low-Level Virtual Machine compiler) [27] instead of OpenCL code, the authors of [28] have taken advantage of the characteristics of this representation to improve the accuracy in the device mapping problem using a RNNs based model. The work in [29] introduced the use of Convolutional Neural Networks (CNNs) instead of Recurrent Neural Networks (RNNs).

3.3 Graph representation

In IR2VEC [7], the authors were the first to propose using Knowledge Graph Embedding (KGE) in code optimization to learn code representation using the TransE model. This representation was suitable input to the XGboost classifier used for the classification. They evaluated the model in two tasks (device mapping and thread coarsening), and its results are considered state-of-the-art in these problems. Using Graph Neural Network (GNN), which has recently become a popular area of research [30] and working on the device mapping and thread coarsening problems, the authors of [31] have tried to use Abstract Syntax Trees (ASTs) and Control Data-Flow Graphs (CDFGs) as a code representation instead of code sequence to do the classification. In the same direction, the authors of [32] proposed a graph-based program representation called Program Graphs for Machine Learning (PROGRAML) that combines the call graph, control-flow graph, and data-flow graph. They

evaluated the model on device mapping and algorithm classification tasks, and it is considered state-of-the-art in the algorithm classification problem.

3.4 Which representation is better

Which representation is better? For example, the work in [29] showed CNN's ability to get interesting results with a sequence of tokens, even better than many models that relied on pre-trained techniques or complex graph representation, using the small dataset of the device mapping problem only.

3.5 Which model is better

Another question arises, Which model is better? And it is a complex question to answer [11]. On a hand, Deep Learning provides a way to generate program features automatically, but it requires a large amount of training data to get accurate results (lack of labeled data is a well-known problem in this field). On the other hand, traditional Machine Learning provides better results in small datasets but relies on the quality of features. The extraction of high-quality features is the main problem here.

4 Proposed approach

In this section, we present our approach. We use a Deep Learning model composed of an embedding layer to learn the representation of code and a CNN layer to extract low-dimension features vector, and then we search for the best classical Machine Learning algorithm to map these features to a particular output to perform the required task.

The previous work is based on features engineering or end-to-end Deep Learning models and attempts to address the issue of data scarcity by introducing more complex representations or employing pre-trained techniques. On the contrary, in our approach, the goal is to avoid feature engineering by using CNN to extract a small number of high-quality features while leveraging the power of traditional methods and techniques developed to deal with small data to address the issue of data scarcity.

Algorithm 1 shows all the steps needed to build our model. The primary input for the model is the intermediate representation of the LLVM compiler (LLVM-IR), like all the recent work. After the preprocessing, a Deep learning model (feature extractor) composed of Embedding, CNN, and Max pooling layers is used to extract features from the LLVM-IR code (we can add auxiliary inputs if needed), then these features are used to train any traditional machine learning algorithm on the downstream tasks, detailed architecture of this model are in Figure 2 (C). To build this model, we need the three stages shown in Figure 2: A) Preprocessing of the LLVM-IR code; B) Training a CNN model; and C) Training a Machine Learning model using the feature extractor built based on the embedding and CNN layers from the CNN model.

4.1 Preprocessing

We preprocessed the LLVM-IR code to convert it to a vector of numbers, which will pass as input to the Deep Learning model. The preprocessing is done by following these steps: (1) Tokenizing the code, and (2) Atomizing the code. An example of the preprocessing stage is illustrated in Figure 3.

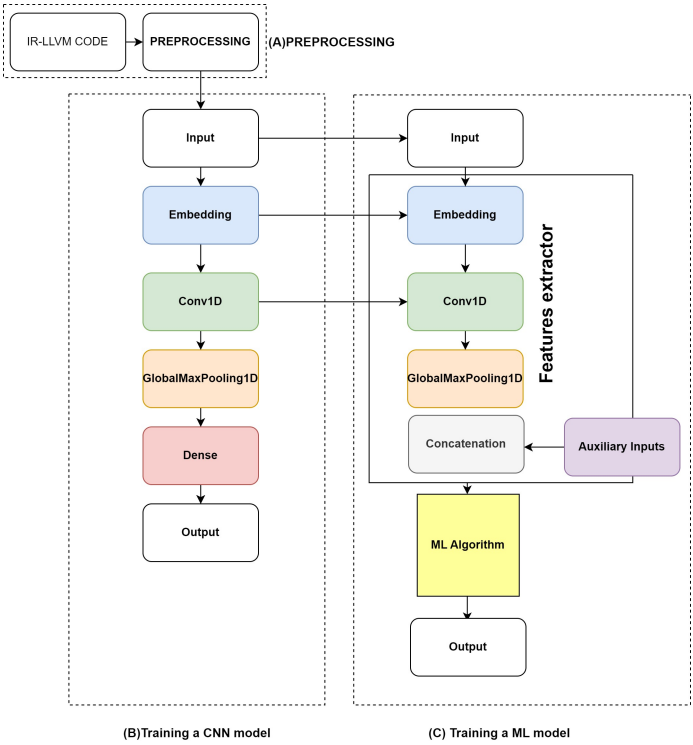

Fig. 2 System Overview: First train a CNN model, then build a Feature extractor model by transfer learning, and finally train a Machine Learning algorithm using the output of the Feature extractor as input.

4.1.1 Tokenization

Before identifying the elements or the tokens in the sequence of code, we rewrite the code by:

- Removing the empty line, comments, and unnamed metadata starting with (#,!);
- Rewriting data types as a single tokens.(E.g. ¡4 * float¿ → 4_float)
- Rewriting align ¡Alignment¿ as a single token.(E.g. align 4 → align_4)
- Replacing the vectors, arrays, and constants (float, integer, double) with a placeholder respecting the types. (E.g. 0x3FD3333340000000

Algorithm 1 The Algorithm of Building Predictive Model:

Input: IR-LLVM code dataset
Output: A predictive model for a device mapping problem

procedure BUILD_OUR_PREDICTIVE_MODEL(*dataset.code*)
▷ Define preprocessing function
  procedure PREPROCESSING(*code*)
    code = Tokenization (code)
    code = Atomization (code)
    return code
  end procedure
▷ Build a CNN model
  procedure CNN_MODEL(*input*)
    x = Embedding(input)
    x= Conv1d(x)
    x = GlobalMaxpooling1D(x)
    Output = Dense (x)
  end procedure
▷ Build a Feature extractor model based on CNN model's layers (Embedding and Conv1d)
  procedure FEATURE_EXTRACTOR_MODEL(*input*, *auxiliary_inputs*)
    x = CNN_model. Embedding(input)
    x = CNN_model. Conv1d(x)
    x= GlobalMaxpooling1D(x)
    Features_vector = Concatenate (x, auxiliary_inputs)
  end procedure
▷ Build Our predictive model
  procedure OUR_PREDICTIVE_MODEL(*input*)
    Features_vector = Feature_extractor_Model (input)
    Prediction = ML (Features_vector)
  end procedure
▷ Preprocess each code in the training dataset
  for each *code* ∈ *Tdataset.code* **do**
    dataset.code = Preprocessing (code)
  end for
▷ Train the CNN model on the training dataset
  Train (CNN_model(Tdataset.code), dataset.labels)
▷ Extract features vector for each code in training dataset
  Dataset.features = Feature_extractor_Model(dataset.code,dataset.
auxiliary_inputs)
▷ Select a machine learning algorithm
  ML = Select_Machine_learning_algorithm ()
▷ Train the ML on the same
  Train (ML, Dataset.features, labels)
  Return Our_predictive_Model()
end procedure

IR-LLVM CODE

```
%20 = fdiv float 1.000000e+00, %19, !fpmath !12
%21 = insertelement <4 x float> undef, float %20, i32 0
%22 = shufflevector <4 x float> %21, <4 x float> undef, <4 x i32> zeroinitializer
%23 = fmul <4 x float> %22, %18
%24 = tail call <4 x float> @_Z4sqrtDv4_f(<4 x float> %23) #4
%25 = fmul <4 x float> %24, <float 0x3FD3333340000000, float 0x3FD3333340000000, float 0x3FD3333340000000, float 0x3FD3333340000000>
```

(1) Tokenization

```
_local  =  fdiv float _float_constant  ,  _local
_local  =  insertelement 4_float undef ,  float _local ,  i32 _int_constant
_local  =  shufflevector 4_float _local ,  4_float undef ,  4_i32 zeroinitializer
_local  =  fmul 4_float _local ,  _local
_local  =  tail call 4_float _function ( 4_float _local )
_local  =  fmul 4_float _local ,    _vectorC_float
```

(2) Atomization

```
1, 5, 54, 11, 16, 2, 1,
1, 5, 85, 62, 74, 2, 11, 1, 2, 21, 4,
1, 5, 113, 62, 1, 2, 62, 74, 2, 82, 112,
1, 5, 25, 62, 1, 2, 1,
1, 5, 23, 20, 62, 19, 18, 62, 1, 17,
1, 5, 25, 62, 1, 2, 96
```

Fig. 3 Example of transformation of an intermediate representation to a sequence of numbers.

→ float_constant, 23 → int-constant, ¡float 1.000000e+01, float 1.000000e+01¿→ vector_float_constant);

- Replacing global and local identifiers, variable, and function names with a placeholder.
- Replacing strings in the declaration of global variables, functions, and string constants by a placeholder (E.g. c"Hello World
  0A
  00", section "foo") This Tokenization is similar to that of [29] except in rewriting alignment and replacing strings, our goal is to reduce the number of tokens vocabulary used as much as possible.

4.1.2 Atomization

In this step, first, we create a dictionary to convert tokens to integers. All code tokens are converted to their integer indexes using this dictionary. Secondly, we get a sequence of integers representing each code.

4.2 Training a CNN model

Figure 2 (B) shows the architecture of our CNN model used in this stage, we use the same architecture of the language model in [29].

4.2.1 CNN model architecture

This model consists of four layers:

1. Embedding layer: An embedding layer that takes a sequence of integers as input and learns a useful representation of it by translating each integer (index of token) into a vector. We use an Embedding size = 64 and choose an input size = 4096 in device mapping and algorithm classification problems and 2048 in thread coarsening problem.
2. One-dimensional convolutional neural network: A CNN that takes the distributed representation as input and applies 1D convolution over it using several filters. Each filter has a kernel size that defines the number of tokens to consider as the convolution passed across the input. We use 64 filters with kernel size = 32 and LeakyReLU as an activation function.
3. GlobalMaxpooling1D: A max-pooling layer that selects the maximum value of each output of filters. The output of this layer is a vector of size of 64.
4. Dense: A fully connected layer with two outputs and "Sigmoid" activation to do the classification.

2) Training: The training was performed for 40 epochs using the Adam optimizer with its default parameters on the training dataset.

4.3 Training a Machine Learning model

Figure 2 (C) shows the architecture of our classifier, we build another CNN model as a Feature extractor with the transfer learning technique, and the

output of this model concatenated with auxiliary inputs (optionally) is used as inputs to train a Machine Learning algorithm.

1. Feature extractor: We built this model using the Embedding and the Conv1D layers from the previous CNN model, a GlobalMaxpooling1D layer, and a Concatenation layer that is used to add the Auxiliary inputs (optionally) to the output. For example, in device mapping problem, the result is a vector with 66 values, 64 from the GlobalMaxpooling1D output + 2 Auxiliary inputs. We used it as a feature extractor that takes a sequence of integers and auxiliary-inputs as input and produces a features vector as output without additional training.
2. Auxiliary Inputs: The auxiliary inputs added to feature vectors are values used in the previous work in the device mapping problem to augment the source code input. Code optimization depends on many variables, including the hardware variables.
3. Algorithms: Once we get a features vector with a small number of dimensions, we can train any Machine Learning algorithm to do the downstream tasks and take advantage of its ability to deal with a small dataset. In this work, we compare seven ML algorithms, including an Ensemble learning for every three downstream tasks, and compare their results. The algorithms and their parameters are as follows: Xgboost: max_depth = 6, learning_rate = 0.1, n_estimators = 100, n_jobs =10; Random forest: max_depth = 5, n_estimators = 50; SVM: C = 1.0, kernel = 'linear', degree = 7, probability = False, gamma = 'auto'; AdaBoostGuassianNB, LR: Default parameters; Section 5 shows the results of the comparison and discusses them.
4. Ensemble Learning: Ensemble learning is one of the most promising techniques to deal with small and imbalanced datasets [33]. This technique uses a set of machine learning algorithms for the classification, then uses their results to get the final result. Since we can use traditional Machine Learning with our feature extractor, we benefit from this technique. In our work, we use StackingClassifier, where the output of each ML algorithm (classifier) in the ensemble is stacked, and another classifier (Meta-classifier) computes the final prediction from this stack. Our ensemble learning is a set of the best three algorithms for each task as classifiers and a meta-classifier that takes the results of these algorithms and produces the final result.

5 Evaluations

We experimentally evaluate the performance of our model on three different downstream tasks (device mapping, optimal thread coarsening, and algorithm classification), comparing it with state-of-the-art models. In this section, we report the results of our experiments.

Table 1 Comparison of ML algorithms accuracy using the Feature extractor(Device mapping)

Algorithms \ Datasets	AMD Tahiti	Nvidia	Average
XG boost	90.58%	88.19%	89.38%
Random forest	89.53%	82.81%	86.17%
Ada Boost	89.83%	85.20%	87.51%
SVM	70.85%	58.59%	64.72%
Gaussian NB	64.12%	53.36%	58.74%
Logistic Regression	40.95%	57.24%	49.10%
Ensemble Learning	92.2%	91.0%	91.6%

5.1 Device mapping

5.1.1 dataset

- For this task, we use an LLVM-IR code dataset provided by [5] and used by the previous work in [29], [31] and [7]. This dataset is the same data set of OpenCL code used by [4] converted to LLVM-IR code.
- The dataset is composed of 680 data for each of the two GPU devices (NVIDIA GTX 970 and AMD Tahiti 7970) and was obtained by executing 256 unique kernels on the CPU and GPU with different workgroups sizes and byte transfer values. These kernels are from several benchmark suites comprising AMD SDK, NPB, NVIDIA SDK, Parboil, PolybenchGpu, Rodinia, and SHOC. Each data point is a kernel converted to LLVM-IR code, the

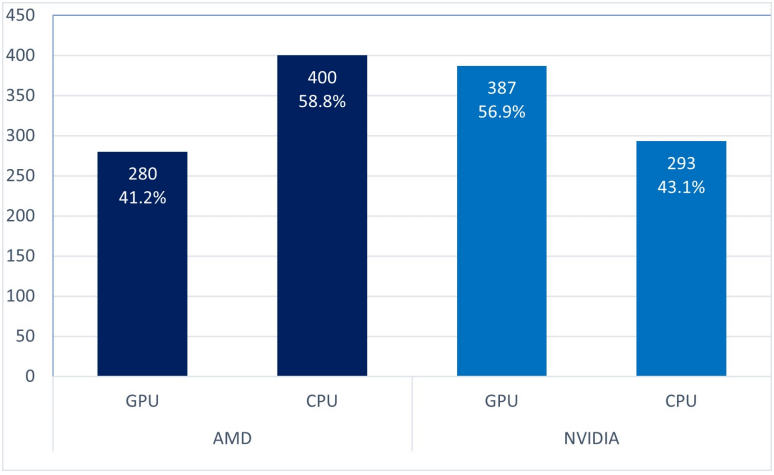

Fig. 4 Classes distribution in the AMD and NVIDIA datasets.

Table 2 Comparison of the accuracy with the state-of-the-art models(Device mapping)

Models \ Datasets	AMD Tahiti	Nvidia	Average
Grewe et al.	73.38%	72.94%	73.16%
DeepTune	83.67%	80.29%	81.98%
Inst2vec (NCC)	82.79%	81.76%	82.27%
ProGraML	86.6%	80.0%	83.3%
DeepLLVM	85.60%	85.32%	85.46%
IR2VEC	92.82%	89.68%	91.25%
Our System	92.2%	91.0%	91.6%

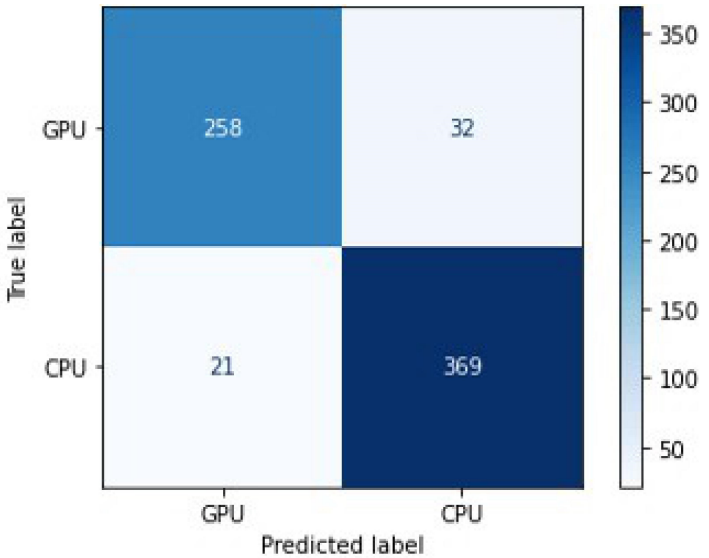

Fig. 5 The confusion matrix on the AMD dataset (device mapping problem using the Ensemble learning)

optimal target device CPU or GPU, and the parameters (workgroup size, byte transfer). Figure 4 shows the distribution of classes in the AMD and NVIDIA datasets.

5.1.2 Training and metrics

We use the accuracy metric with 10-fold cross-validation to compare our system to previous work. The data set is split randomly into 10 subsets, and we train 10 classifiers by changing the subset used as a test set. Initially, we train the CNN model on the training set, and then we build the Feature extractor

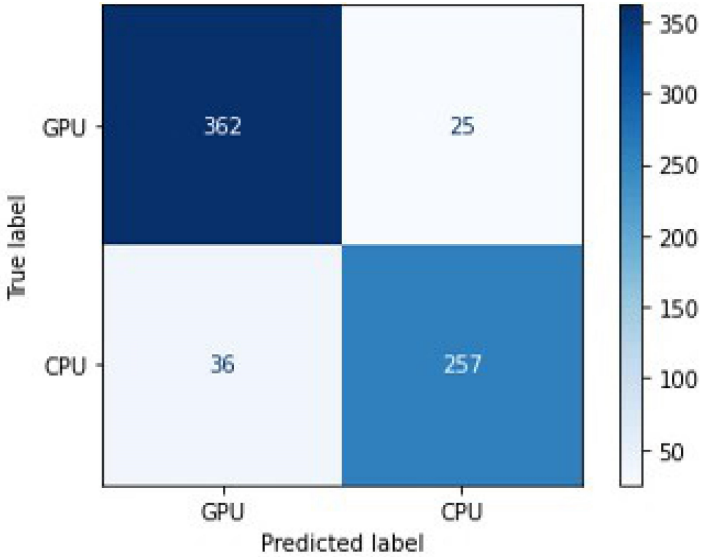

Fig. 6 The confusion matrix on the NVIDIA dataset.(device mapping problem using the Ensemble learning)

model by transfer learning. Finally, we train the Machine Learning algorithm on the same training set using the Feature extractor output as input. The predictions were made using the model shown in Figure 2 (C) on the test dataset to compute the accuracy and other metrics by taking the average of the 10 classifiers.

5.1.3 Machine learning algorithms comparison

In this task, we aimed to evaluate the accuracy of seven Machine Learning algorithms, including Ensemble learning. The results, presented in Table 1, demonstrated that tree-based algorithms using the ensemble technique (Xgboost,

Table 3 Comparison of Precision,Recall,and F1 score, with the state-of-the-art models(Device mapping problem on AMD dataset).

Metrics / Models	Precision	Recall	F1
DeepTune	0.72	0.72	0.72
Inst2vec (NCC)	0.81	0.80	0.80
ProGraML	0.89	0.87	0.88
DeepLLVM	0.83	0.81	0.82
IR2VEC	0.92	0.87	0.89
Our System	0,92	0,88	0,90

Random Forest, Adaboost) outperformed the others. This finding was not surprising given the small dataset used. Moreover, we developed our ensemble learning based on these three algorithms, resulting in the highest accuracy.

5.1.4 Comparison with previous work

First, we compare the accuracy of our system with the previous work. Table 2 shows the results of this comparison. Our model produced almost the same accuracy (92.2%) compared to the state-of-the-art model IR2VEC (92.82%) on the AMD dataset while providing better accuracy (91.0% compared to 89.68%) than it on the Nvidia dataset. Our model outperforms the performances of all the rest models for both datasets, despite the use of only the small dataset of device mapping and without trying to add any other information such as control and data flow. Second, we compare our system with previous work on other metrics (Precision, Recall, F1 score). Figure 5 and Figure 6 present the confusion matrix of our system on the two datasets AMD and NVIDIA respectively, used to calculate the previously mentioned metrics. Table 3 showcases the results of our system and previous models for metrics such as Precision, Recall, and F1 score on the AMD dataset. Similarly, Table 4 provides these metric values for the NVIDIA dataset. We have obtained the metrics values from [32] for most models, except for IR2VEC and DeepLLVM, where we replicated their experiments to calculate these metrics since they were not available in the literature. Our system demonstrates superior performance across all metrics (accuracy, precision, recall, and F1) compared to previous models in both datasets.

Based on our findings, our model consistently attains superior results to the most advanced models in this particular task using a small amount of training data and without any reliance on supplementary information, complex representation, or pre-training on an extensive dataset.

Table 4 Comparison of Precision,Recall,and F1 score, with the state-of-the-art models(Device mapping problem on NVIDIA dataset).

Metrics / Models	Precision	Recall	F1
DeepTune	0.69	0.61	0.65
Inst2vec (NCC)	0.79	0.79	0.79
ProGraML	0.81	0.80	0.80
DeepLLVM	0.81	0.83	0.82
IR2VEC	0.85	0.88	0.87
Our System	0,90	0,93	0,92

Table 5 Comparison of ML algorithms accuracy using the Feature extractor (Algorithm classification)

Metric / Algorithms	Accuracy
End to End DL	93.32%
XG boost	93.92%
Random forest	90.69%
Ada Boost	93.61%
SVM	95.47%
Gaussian NB	86.92%
Logistic Regression	93.89%
Ensemble Learning	92.27%

Table 6 Comparison of the accuracy with the state-of-the-art models (Algorithm classification)

Metric / Models	Accuracy
TBCNN	94.00%
NCC (inst2vec)	94.83%
ProGraML GGN	96.32%
ProGraML Transformer	96.67%
Our model	95.48%

5.2 Algorithm classification

5.2.1 dataset

For this task, we use the dataset used in previous work [5], [32]. This dataset is the POJ-104 [16] dataset converted to LLVM-IR representation. This dataset contains 104 program classes and around 500 samples for each class split in 3:1:1 for training, validation, and testing. Data augmentation was applied by the authors of [5] on the training set, by compiling each program with different flags. The dataset (after the data augmentation) is composed of around 220K training data and around 10k data for each validation and test data.

5.2.2 Training and metrics

Our approach for this task is to train our CNN model utilizing the same parameters as we did for the device mapping problem. In line with our previous experiments, we constructed the Feature extractor and trained the ML algorithms accordingly. The accuracy metric was employed to compare the performance of various ML algorithms and our model with the previous work. To ensure a more dependable evaluation, we conducted the experiment five times and calculated the average accuracy from the five runs

5.2.3 Machine learning algorithms comparison

As we can see in the results in Table 5.1.4, despite using a relatively large dataset, traditional algorithms, including logistic regression, perform better than the end-to-end Deep Learning CNN model, this was an unexpected outcome. Additionally, while ensemble learning excelled in the device mapping problem, it underperformed compared to end-to-end Deep Learning in this experiment. Notably, the Support Vector Machine achieved the highest accuracy among all models tested, outperforming tree-based algorithms used within an ensemble technique. These findings demonstrate the importance of tailoring models to specific datasets or problems based on experimental outcomes.

5.2.4 Comparison with previous work

We conducted a comparison between our model and state-of-the-art models including TBCNN [16], inst2vec [5], and ProGraML [32] in this task. The findings are presented in Table 6. Our results reveal that our model surpasses the performance of the first two models and achieves comparable results (95.47%) to ProGraML (96.67%). It is noteworthy that while our model does not rely on pre-trained techniques like Inst2vec or complex graph-based code representations involving control and data flow information like PrograML, it still demonstrates advantageous outcomes. Furthermore, even with a reduced training dataset size, approximately 30% or around 86 data points per class, our model exhibits consistently high accuracy levels above 95%. In contrast, for end-to-end deep learning CNN models any decrease in training data leads to decreased accuracy.

5.3 Thread coarsening

In OpenCL, the process of merging parallel threads into a single one is called Thread-coarsening. It is controlled by the thread coarsening factor, which determines how many threads to combine. Choosing the optimal value for this parameter is vital to achieving optimal performance in a kernel for a specific architecture. However, it's worth mentioning that what may work well for one architecture may not necessarily work for another, and using the same factor could result in reduced efficiency.

5.3.1 dataset

- For this task we used the dataset employed in [5] and [7], this dataset is the dataset used in the previous work on the thread coarsening problem including [4] converted from OpenCL code to LLVM-IR code.
- The dataset is composed of 68 data points obtained by executing 17 OpenCL kernels (taken from AMD SDK, NVIDIA SDK, and Parboil benchmarks suites) on 4 different GPU devices AMD Radeon 5900, AMD Tahiti 7970, NVIDIA GTX 480, and NVIDIA Tesla K20c, with all the 6 classes of possible coarsening factors, to determine the best coarsening factor for each kernel on each device [4], [21], [32], [13] [33]. each data is composed of kernel converted to LLVM-IR code, GPU device, and the optimal thread coarsening.

5.3.2 Training and metrics

In this task, we used the same parameters for training our CNN model as in the two previous tasks. However, we made one change by decreasing the input size to 2048 due to the longest kernel in the preprocessed dataset being 2043. To evaluate our models, we employed leave-one-out cross-validation. This method involved training a model on 16 kernels from 17 for each device and using it to predict the coarsening factor of the excluded kernel. The leave-one-out cross-validation is used in all previous work [5], [4], [31], [6] except the work in [7], in which they used k-fold cross-validation on the entire dataset, where the model was trained on 67 kernels and used to predict the best thread coarsening factor of the excluded kernel. We use the speedup metric and aggregate the results with the geometric mean for evaluation and comparison. This same approach that used in [31] and [7] instead of the arithmetic mean used in the earlier work [4], [31], [6]. The two work [31] and [7] reproduced the other models and their results using the geometric mean, and the results obtained by the two work are different because of the difference in the configuration of the evaluation (leave-one-out cross-validation and k-fold cross-validation). We reproduced all previous work and their results for this task to ensure a fair comparison.

5.3.3 Machine learning algorithms comparison

Like in the previous tasks, we have compared the End-to-End deep learning CNN model, the seven ML algorithms, and the ensemble learning (built based on the best three algorithms). As we can see in Figure 7, six ML algorithms out of seven perform better than the End-to-End Deep Learning and give a speedup (geometric mean across all platforms) as opposed to the End-to-End DL model, which yields an overall slowdown (speed up of around 0.98). In our experiments, the Random Forest algorithm produced the best results for this task with an overall speedup of 1.05. We note again in this task that another different algorithm gives the best result compared to the first two tasks.

5.3.4 Comparison with previous work

In comparison to other models, we can see in Figure 8 that our model (features extractor + random forest) demonstrated the highest speedup of 1.05x (geometric mean across all platforms). This outperforms the speedups of 1.02x achieved by IR2vec [7] and 1.01x achieved by Inst2vec [5]. Conversely, the remaining models, including the work in [6], DeepTune [4], and GNN-AST [31], showed an overall slowdown. Figure 9 shows the speedups obtained by all models (geometric mean overall executions) for each platform. As we can see on AMD Radeon, IR2vec achieved a speedup of 1.16x compared to the 1.15

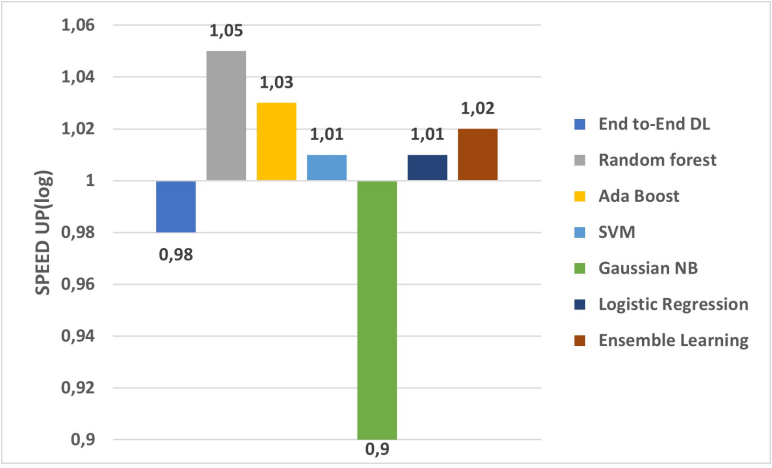

Fig. 7 Speedups (geometric mean across all platforms) comparison between different ML algorithms(Thread coarsening)

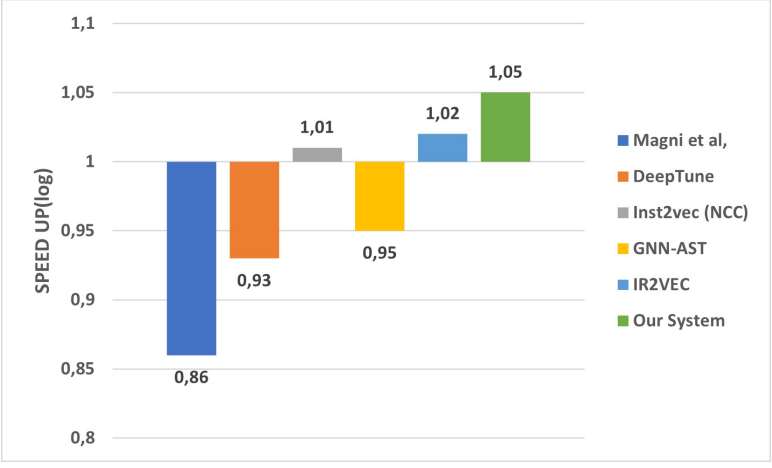

Fig. 8 Speedup (geometric mean overall platforms) compared to state-of-the-art models (Thread coarsening)

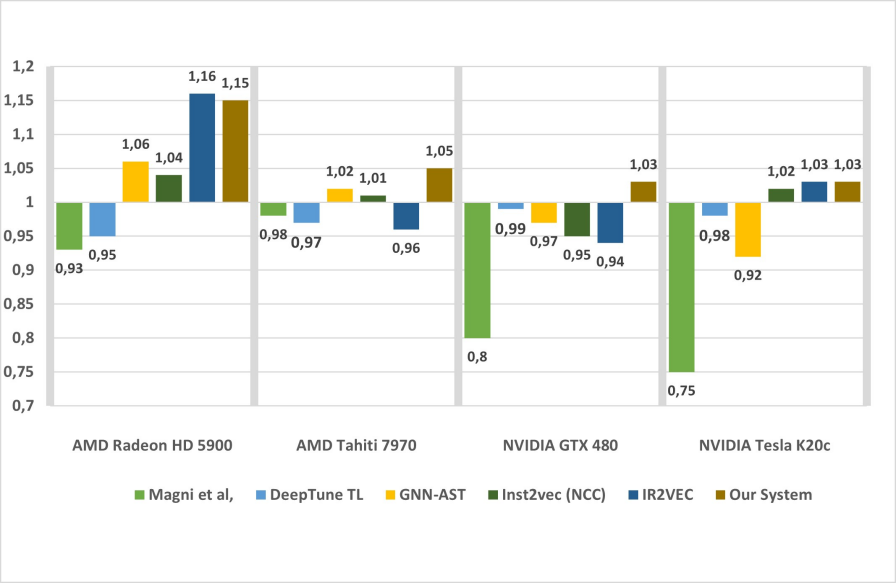

Fig. 9 Speedup (geometric mean for each platform) compared to state-of-the-art models (Thread coarsening)

a by our model, and this is the only platform where we don't get the best result. GNN-AST and Inst2vec achieved a speedup of 1.06x and 1.04x, while the other models yielded a slowdown. On AMD Tahiti, we achieved a speedup of 1.05 compared to 1.02x and 1.01x achieved by GNN-AST and Inst2vec, respectively, while the other models resulted in a slowdown, including IR2vec. On the Nvidia GTX, we are the only ones who get a speedup (1.03x), While all other models yielded a slowdown. On NVIDIA Tesla, we achieved a speedup of 1.03x the same get by IR2vec. Inst2vec achieved a speedup of 1.02x, and all other models yielded a slowdown.

In conclusion, our model outperforms state-of-the-art models when considering overall executions across all platforms. Specifically, we achieve the best results on two out of four platforms, and match the performance of the IR2vec model, on another platform. While IR2vec outperforms our model by a mere 0.01 on one platform, we are the first to achieve a positive speedup on all four platforms.

6 Conclusion and future work

We have presented a novel approach to tackle code and compiler optimization problems. Using a Convolutional Neural Network, we extract features for input into traditional Machine Learning algorithms. This approach enables us to efficiently search for the most suitable algorithm for specific tasks and leverage established techniques, such as ensemble learning to handle challenges associated with small datasets. Our results demonstrate that our simple approach

surpasses pre-trained models and complex graph-based representations, leading to reduced training time requirements and decreased data needs. The proposed model exhibited improved performance compared to existing models in two tasks: device mapping and thread coarsening while achieving comparable results in algorithm classification. Although our approach performs well, it uses a simple code representation that may not capture the complete intricacy of the code. More efficient techniques, such as graph-based representations, have demonstrated their ability to capture the structural information of code. However, an extensive dataset is required to learn a meaningful code representation.

In our future research, we intend to further evaluate the effectiveness of our proposed approach in more tasks. Furthermore, we strive to enhance the performance of both the model and feature extractor by exploring methods that leverage information from more intricate representations while working with limited data.

References

[1] Ghazi, A.N., Petersen, K., Börstler, J.: Heterogeneous systems testing techniques: An exploratory survey. In: International Conference on Software Quality, pp. 67–85 (2015). Springer

[2] Chen, C., Fang, J., Tang, T., Yang, C.: Lu factorization on heterogeneous systems: an energy-efficient approach towards high performance. Computing **99**(8), 791–811 (2017)

[3] Singh, S.: Computing without processors: Heterogeneous systems allow us to target our programming to the appropriate environment. Queue **9**(6), 50–63 (2011)

[4] Cummins, C., Petoumenos, P., Wang, Z., Leather, H.: End-to-end deep learning of optimization heuristics. In: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 219–232 (2017). IEEE

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

[6] Magni, A., Dubach, C., O'Boyle, M.: Automatic optimization of thread-coarsening for graphics processors. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 455–466 (2014)

[7] VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M.S., Upadrasta, R., Srikant, Y.: Ir2vec: Llvm ir based scalable program embeddings. ACM

Transactions on Architecture and Code Optimization (TACO) **17**(4), 1–27 (2020)

[8] Grewe, D., Wang, Z., O'Boyle, M.F.: Portable mapping of data parallel programs to opencl for heterogeneous systems. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–10 (2013). IEEE

[9] Ashouri, A.H., Killian, W., Cavazos, J., Palermo, G., Silvano, C.: A survey on compiler autotuning using machine learning. ACM Computing Surveys (CSUR) **51**(5), 1–42 (2018)

[10] Memeti, S., Pllana, S., Binotto, A., Kołodziej, J., Brandic, I.: Using meta-heuristics and machine learning for software optimization of parallel computing systems: a systematic literature review. Computing **101**(8), 893–936 (2019)

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

[12] Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR) **51**(4), 1–37 (2018)

[13] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al.: Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)

[14] Fang, J., Huang, C., Tang, T., Wang, Z.: Parallel programming models for heterogeneous many-cores: a comprehensive survey. CCF Transactions on High Performance Computing **2**(4), 382–400 (2020)

[15] Czarnul, P., Proficz, J., Drypczewski, K.: Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems. Scientific Programming **2020** (2020)

[16] Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, pp. 1287–1293 (2016)

[17] Deniz, E., Sen, A.: Using machine learning techniques to detect parallel patterns of multi-threaded applications. International Journal of Parallel Programming **44**(4), 867–900 (2016)

[18] Baghdadi, R., Merouani, M., Leghettas, M.-H., Abdous, K., Arbaoui, T.,

Benatchba, K., Amarasinghe, S.P.: A deep learning based cost model for automatic code optimization. ArXiv **abs/2104.04955** (2021)

[19] Wang, Z., Tournavitis, G., Franke, B., O'boyle, M.F.: Integrating profile-driven parallelism detection and machine-learning-based mapping. ACM Transactions on Architecture and Code Optimization (TACO) **11**(1), 1–26 (2014)

[20] Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: International Symposium on Code Generation and Optimization, pp. 123–134 (2005). IEEE

[21] Wen, Y., Wang, Z., O'boyle, M.F.: Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In: 2014 21st International Conference on High Performance Computing (HiPC), pp. 1–10 (2014). IEEE

[22] Luk, C., Hong, S., Qilin, K.: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42), pp. 45–55

[23] Liu, B., Zhao, Y., Zhong, X., Liang, Z., Feng, B.: A novel thread partitioning approach based on machine learning for speculative multithreading. In: 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, pp. 826–836 (2013). IEEE

[24] Wang, Z., O'boyle, M.F.: Using machine learning to partition streaming programs. ACM Transactions on Architecture and Code Optimization (TACO) **10**(3), 1–25 (2013)

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

[26] Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages **3** (2018). https://doi.org/10.1145/3290353

[27] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86 (2004). IEEE

[28] Barchi, F., Urgese, G., Macii, E., Acquaviva, A.: Code mapping in heterogeneous platforms using deep learning and llvm-ir. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6 (2019).

IEEE

[29] Barchi, F., Parisi, E., Urgese, G., Ficarra, E., Acquaviva, A.: Exploration of convolutional neural network models for source code classification. Engineering Applications of Artificial Intelligence **97**, 104075 (2021)

[30] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.: A comprehensive survey on graph neural networks (2019)

[31] Brauckmann, A., Goens, A., Ertel, S., Castrillon, J.: Compiler-based graph representations for deep learning models of code. In: Proceedings of the 29th International Conference on Compiler Construction, pp. 201–211 (2020)

[32] Cummins, C., Fisches, Z.V., Ben-Nun, T., Hoefler, T., Leather, H.: Programl: Graph-based deep learning for program optimization and analysis. arXiv preprint arXiv:2003.10536 (2020)

[33] Dong, X., Yu, Z., Cao, W., Shi, Y., Ma, Q.: A survey on ensemble learning. Frontiers of Computer Science **14**(2), 241–258 (2020)