IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received July 15, 2018, accepted August 18, 2018, date of publication August 24, 2018, date of current version September 21, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2867151

Using Ensemble Learning to Improve Automatic Vectorization of Tensor Contraction Program

HUI LIU[1,2], RONGCAI ZHAO[1], AND KAI NIE[1,3], (Member, IEEE)
[1]State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Information Engineering University, Zhengzhou 450001, China
[2]College of Computer and Information Engineering, Henan Normal University, Xinxiang 453007, China
[3]School of Information Engineering, Zhengzhou University, Zhengzhou 450001, China

Corresponding author: Hui Liu (liuhui806@126.com)

This work was supported in part by the National Key Research and Development Program High-Performance Computing Key Special under Grant 2016YFB0200503 and in part by the Henan Normal University Youth Science Foundation under Grant 2015QK21.

ABSTRACT Automatic vectorization is crucial for improving the performance of computationally intensive programs. Existing compilers use conservative optimization strategies for automatic vectorization, which, in many cases, lead to the loss of vectorization opportunity. Studies have shown that the use of machine learning algorithms to build a performance prediction model is beneficial to improve the program performance. The model input is program features, and the output is the predicted optimization strategies or the program performance related to the optimization. In this paper, we focus on a computational intensive loop structure-tensor contraction, which is common in quantum chemical simulations. Most existing machine learning methods rely on control and data flow graphs as features to represent programs, but different tensor contraction kernels have the same control and data flow graphs. In addition, the existing methods often use the same kind of learning algorithm to construct a learning model, which is prone to overfitting and low-precision problems. In this paper, we propose an automatic vectorization performance enhancement method based on ensemble learning. We construct an ensemble learning model to predict the performance of tensor contraction kernels with different vectorization strategies and select the best vectorization strategy for the kernels. According to the storage access patterns of the tensor contraction kernels, we propose a static method for features representation. Based on the multi-algorithm ensemble learning strategy, we obtain better learning results than the single learning algorithm. The experimental results show that the prediction model achieves 88% and 87% prediction efficiency on two different platforms with different instruction sets, data types, and compilers. Compared with the existing methods, the prediction efficiency is greatly improved. In addition, the average peak performance is 2.96× of Intel ICC 12.0 and 2.98× of GCC 4.6 compiler, respectively.

INDEX TERMS Automatic vectorization, compiler optimization, ensemble learning, program features.

I. INTRODUCTION

With the rapid development of multimedia industry, multimedia extended instruction sets are integrated in the processor by the manufactures. The instruction sets use Single Instruction Multiple Data (SIMD) extension technology, which can simultaneously operate multiple data with a single instruction. SIMD extension components were originally only used in the field of multimedia and digital signal processor. Researchers subsequently applied SIMD extension components to high performance computer, such as IBM BlueGene/L and Sunway BlueLight MPP supercomputers which are integrated with short vector extension components. The SIMD extension instructions can load the data with continuous memory address into the vector register at one time, and realize the parallel processing for all data elements in the SIMD vector register. Data need to be loaded multiple times when without the SIMD. With the increasing degree of SIMD parallelism in processors, the effectiveness of automatic vectorization in compilers is crucial. The existing mainstream compilers, such as Open64, ICC and GCC, have already integrated automatic vectorization. At the same time, manual vectorization program is not only easy to make mistakes, but also need the programmers to understand the SIMD instruction sets, so automatic vectorization becomes the primary choice for SIMD parallelism exploration.

Automatic vectorization of existing commercial compilers usually produce better performance compared with the scalar code. However, the vectorization code performance achieved is usually much lower than the peak performance of the processors, even when the data fits within L1 cache and no cache misses are incurred. The main reason for the low performance of the vectorization code generated by the current compilers is that there are a lot of possible program transformations from high-level nested loop to assembly code, so it is very difficult to select the optimal transformation efficiently. The complex execution pipeline in modern processors makes it challenging to develop performance analysis models to predict execution time of machine instruction sequences. Therefore, commercial compilers usually use simple heuristic rules to guide loop transformations and code generation.

In this paper, we focus on Tensor Contraction kernels (TC), which are very common computational intensive loop structures in quantum chemical simulations or molecular folding simulations. Tensor can be seen as a further extension of the matrix. Vector is a first order tensor, and matrix is a second order tensor. If a number of same-dimensional matrices stack together to form an array with cube shaped, then the array is a third order tensor. In linear algebra theory, TC is essentially a generalized inner product of high dimensional matrix, where tensors can have more than two dimensions and perform summation operations on multiple dimensions. The mathematical definition of Tensor and TC is described in detail later. Various types of TC are needed in the high-precision quantum chemical models, such as coupled cluster methods [1].

When using the existing commercial compiler to vectorize the TC codes, the conservative optimization strategies are adopted, and the performance realized is usually far lower than the peak performance of the machines. Fig.1 shows a TC code segment. Array A and array B represent the input tensor, and the output tensor is array C. The existing compilers, such as Intel ICC and GNU GCC, can't execute automatic vectorization for this TC code by default. However, we can vectorize this code by manual vectorization. Therefore, how to improve the automatic vectorization performance of the TC kernels is very important to improve the efficiency of quantum chemical simulation programs and so on.

```
float A[64][64][64]
float B[64][64]
float C[64][64][64]
for(i=0; i<N; i++)
  for(j=0;j<M; j++)
    for(k=0; k<K; k++)
      for(l=0; l<L; l++)
          C[l][j][k]+=A[i][j][k]*B[i][l]
```

FIGURE 1. Example of the TC kernel.

There has been some researches in using machine learning to predict the best compiler optimization strategy, such as compiler optimization options [2], [3], effective loop unrolling factor [4] and block size selection [5]. Our work aims at improving the automatic vectorization performance of the TC code based on the machine learning model. The input of our machine learning model is the program features, and the output is the program performance generated by the optimal automatic vectorization strategy, which belongs to the regression problem in machine learning. In the machine learning models, program features are critical to the final results of the predictions. In the existing researches, the extraction of the program features is mostly dependent on the Control and Data Flow Graph (CDFG) or the static assembly instruction counts. But for the TC programs, there are exactly the same CDFG and assembler instruction counts before compiler optimization, only the order of access memory is different. Therefore, the existing feature extraction methods are not suitable for the TC programs. In addition, the existing compiler optimization methods based on the machine learning often use single learning algorithm, which sometimes have drawbacks in model fitting and classification accuracy. Ensemble Learning (EL) is an effective technique to improve the performance of learning. Ensemble learning can be used for the ensemble of classification problem, regression problem, feature selection, outlier detection, etc [6]. Ensemble learning technology can effectively improve the prediction performance by effectively utilizing the diversity of different learning algorithms, and reduce the variance while not increasing the deviation.

Based on ensemble learning technology, we propose an automatic vectorization performance prediction model ELAV. The input of ELAV is the features of TC kernels, and the output is the runtime prediction of the program when using different vectorization strategies. The experimental results show that ELAV can achieve better prediction performance for the new TC kernels. The main contributions are summarized as follows:

- A static feature representation method is proposed, which can be used to represent the storage access mode of the TC kernels. The static features can be obtained without the actual running of the program, and it's easier to embed this feature extraction process into the existing compilation procedure.
- A vectorization performance prediction model based on the ensemble learning is proposed. By integrating the learning results of multiple different algorithms, better learning results are obtained than the single, and the overall prediction accuracy of the model is improved.
- By employing different platforms, instruction sets, data types and compilers, the vectorization performance prediction for TC kernels in Coupled Cluster Single Double (CCSD) are achieved by the ELAV. Compared with the existing methods, the effectiveness and generality of ELAV are verified.

The structure of this paper is as follows: In Section 2, we introduce the related technologies and concepts. In Section 3, we detail the method of ELAV, including the ensemble learning, the program feature extraction method and the vectorization space. In Section 4, we describe the experimental setup and the analysis of experimental results. In section 5, we introduce the related work. We present our conclusions in section 6.

II. BACKGROUND

A. SIMD

SIMD is a parallelization technique similar to vector operations, which can perform the same operations on multiple data elements at the same time [7]. Currently, SIMD extensions components are integrated with most processors. The instruction sets that support the operation of SIMD extension components are becoming more and more perfect and efficient. The mainstream SIMD instruction sets mostly depend on the design and implementation of the underlying hardware.

SIMD extension instructions exploration can be divided into two categories: one is manual vectorization, that is, programmers write vectorized programs manually. The other is automatic vectorization by the compiler directives. Manual vectorization can make full use of SIMD extension components characteristics and maximize the performance of the program. However, the manual vectorization has poor scalability and portability. Efficient vectorization programs not only require the programmers to understand the programs itself, but also need them to fully understand the hardware architectures.

Compared with manual vectorization, automatic vectorization technology has the characteristics of easy to implement, high accuracy, good scalability and portability. Automatic vectorization has gradually become an efficient program optimization technology. The main benefits of SIMD vectorization come from parallel access and calculation of the data. By loading data from memory into a vector register one-time and perform the same operation to multiple data objects at the same time, SIMD vectorization can effectively improve the execution efficiency of the programs. As shown in Fig.2, we take the vector addition of 256 bits SIMD vector expansion components as an example. This components can be used to load or store 4 double precision floating point scalar at

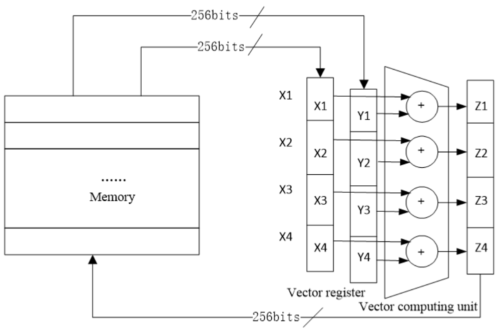


FIGURE 2. Operation diagram of SIMD expansion components.

one-time, and perform addition operations on 4 data elements simultaneously.

In the vector income evaluation of the compiler, many factors need to be considered, such as data access mode, the calculation characteristics in the loop. SIMD load/store operations are mainly to realize data load and store between memory and register. When the data storage address is continuous and aligned according to the width of vector register, the vectorization performance income is higher. When the data storage address continuous but non-aligned, we need to use the non-aligned pack/unpack instructions, which will add additional overheads, and reduce the income of vectorization. In other cases, we must load/store the scalar separately, then pack/unpack it to SIMD register, and this will lead to the lowest performance benefits. However, as the TC kernel shown in Fig.1, at least one of the four loop indexes i, j, k, l belongs to this case.

B. TENSION CONTRACTION KERNELS

Tensor is a higher order extension of vector. In essence, a tensor is a multidimensional array. Each element in the tensor has multiple indicators, each of them represents a model of tensors [8]. For vector spaces $U^{(1)}, U^{(2)}, \ldots, U^{(n)}$, the outer product (denote as $\otimes$) space $U^{(1)} \otimes U^{(2)} \otimes \ldots \otimes U^{(M)}$ is defined as a vector space $\sum ku^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u^{(M)}$ which contains all linear combinations of $u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u^{(M)}$, where $u^{(1)} \in U^{(1)}, u^{(2)} \in U^{(2)}, \ldots, u^{(M)} \in U^{(M)}, k \in R$. $\otimes$ needs to meet the requirement of multi-linear:

$$(\alpha u_1^{(1)} + \beta u_2^{(1)}) \otimes u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u^{(M)} = \alpha u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u^{(M)} + \beta u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u^{(M)} \quad (1)$$

$$u^{(1)} \otimes (\alpha u_2^{(1)} + \beta u_2^{(2)}) \otimes u^{(2)} \otimes \ldots \otimes u^{(M)} = \alpha u^{(1)} \otimes u_1^{(2)} \otimes \ldots \otimes u^{(M)} + \beta u^{(1)} \otimes u_2^{(2)} \otimes \ldots \otimes u^{(M)} \quad (2)$$

$$u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes (\alpha u_1^{(M)} + \beta u_2^{(M)}) = \alpha u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u_1^{(M)} + \beta u^{(1)} \otimes u^{(2)} \otimes \ldots \otimes u_2^{(M)} \quad (3)$$

where $\alpha, \beta \in R, u^{(i)}, u_1^{(i)}, u_2^{(i)} \in U^{(i)}$.

The elements $X \in U^{(1)} \otimes U^{(2)} \otimes \ldots \otimes U^{(M)}$ in this space are defined as ***M* order tensors**. Similar to the matrix, element in the tensor can be represented as $X_{l_1 \times l_2 \times \cdots \times l_M}$, $(1 \leqslant l_i \leqslant L_i, 1 \leqslant i \leqslant M)$, where l_i represents the position of this element in the i-th dimension of the array. If we assume the tensor $X \in R^{L_1 \times L_2 \times \cdots \times L_M}$, $Y \in R^{L_1 \times L_2 \times \cdots \times L_M}$, then the **Tension Contraction** of X and Y is :

$$[X \times Y; (1:M)(1:M)] = \sum_{l_1=1}^{L_1} \cdots \sum_{l_M=1}^{L_M} (X)_{l_1 \times l_2 \times \cdots \times l_M} (Y)_{l_1 \times l_2 \times \cdots \times l_M} \quad (4)$$

The condition of TC is that the tensor X is equal to the tensor Y on a given order (or multiple orders), that is, has equal number of elements in a certain dimension. Every occurring of a TC will reduce the order by 2.

We take the TC kernels in Couple Cluster Singles and Doubles (CCSD) as research objects [8]. The TC kernels in

CCSD consists of perfect nested loops, with a single multiplication cumulative statement in the innermost loop. There are discontinuous and irregular memory access patterns in TC kernels computing statements, which make it difficult to use the existing compilers to perform automatic vectorization. Each index variable of an array in the TC kernels appears 2 times, and each variable is accessed at most once a time. In the TC kernel shown in Fig.1, the dimension size of i, j, k, l is N, M, K and L, respectively. The computational kernel mainly operates on array A, B and C, the dimensions of array A and array C is 3, and array B is 2.

C. ITERATIVE COMPILATION BASED ON MACHINE LEARNING

Iterative Compilation which can integrate different optimization techniques effectively is an optimization method for the general programs. This method generates a series of program versions by mining a variety of program optimizations, and selects a program version with the maximum improvement through executing the different versions on the target platform. The performance optimization effect of iterative compilation is significantly better than the static compilation. However, the selection of transformations, the order and number of the transformations in the iterative compilation optimization process, all result in a huge optimization space. At the same time, iterative compilation is a mechanical search, and lacks of using the experience gained before. In addition, the existing compilers usually use fixed optimizations for programs on a given target platform. But different codes may require a customized optimization to achieve better performance. Especially for the embedded system which is more dependents on the compile-time optimization, because the computer architectures are more limited to memory size, structure and processor speed, so a higher degree of customization is demanded [9].

In recent years, researchers have applied artificial intelligence to compiler optimization heuristics of the iterative compilation [10], [11]. Compiler optimization based on machine learning is a method of predicting program optimization strategies to maximize code performance. Most of the existing compiler optimization strategies rely on the experiences of compiler developers. These optimization rules are made up of very complex codes, and only a few compiler developers can understand the codes. Compiler optimization techniques based on machine learning use a trained machine learning model to replace the original optimization engine of the compiler. The input of learning model is the features of the program, and the output is the program performance or the optimization strategy predicted by the learning model. Our prediction model is based on ensemble learning method, the model input is the features of TC kernel, and the output is program execution time corresponding to different vectorization strategies. We choose the optimal vectorization strategy with the shortest program running time.

III. PROPOSED METHODOLOGY

The main goal of the ELAV model is to construct a vectorization performance prediction model for TC programs. Each TC kernel passes through a feature extraction phase, then the parametric representation of the kernel is generated. Prediction model makes these parameterized expressions associate with automatic vectorization performance of the TC kernels. The vectorization strategy with the maximum performance is the one we should select. This belongs to the regression problem in the machine learning. The model framework is shown in Fig.3. The framework consists of two main stages: model training phase and model using phase. In the model training phase, the ensemble learning based prediction model is trained on the basis of the training set. In the model using phase, the maximum performance corresponds to the best vectorization strategy for the new TC kernels is predicted by the knowledge stored in the prediction model.

In the model training phase, we first extract the static feature *VECfeatures* for the training set TC kernels. *VECfeatures* is a new features set for the TC kernels. Then, generate a vector versions set $\{vec_1, vec_2, \ldots, vec_n\}$ for each TC kernel. Static features *VECfeatures* mainly include the important attributes of the memory access mode, such as data continuity, locality, dependency, and whether there exists reduction operation. These attributes are mainly obtained by analyzing the subscript index of the loop array. $\{vec_1, vec_2, \ldots, vec_n\}$ represents n vectorization versions corresponding to n vectorization options. The vectorization options are shown in Table.1, which conclude whether or not vectorizing each layer in the nested loop. When vectorizing a certain layer, the loop unroll factor can be set to 1,2,4,8,16. For 4-layer nested loop (our TC kernels are up to 4-layer nested loop), we have 21 vectorization options. We run all vectorization versions $\{vec_1, vec_2, \ldots, vec_n\}$ on the target machine, and record the performance $\{per_1, per_2, \ldots, per_n\}$. For a TC kernel in the training set, the tuple $(VECfeatures, per_i)$ $1 \leqslant i \leqslant n$ constitutes n training samples of the ELAV model. The vectorization version corresponding to the maximum performance is the optimal vectorization option for the TC kernel. In order to improve the generalization ability and accuracy of the model, we use standard Leave-one-out Cross Validation (LOOCV) to evaluate the model on the training set. That is, the model is trained on all vectorization versions of the $N-1$ TC kernels, then the model is evaluated on the remaining kernel. In the model using phase, the features of new TC kernel are extracted firstly, then the optimal vectorization performance of TC kernel is predicted by the knowledge stored

TABLE 1. Vectorization strategies.

vectorization	0	1	2	3	4
unroll factor	0	1	1	1	1
	0	2	2	2	2
	0	4	4	4	4
	0	8	8	8	8
	0	16	16	16	16

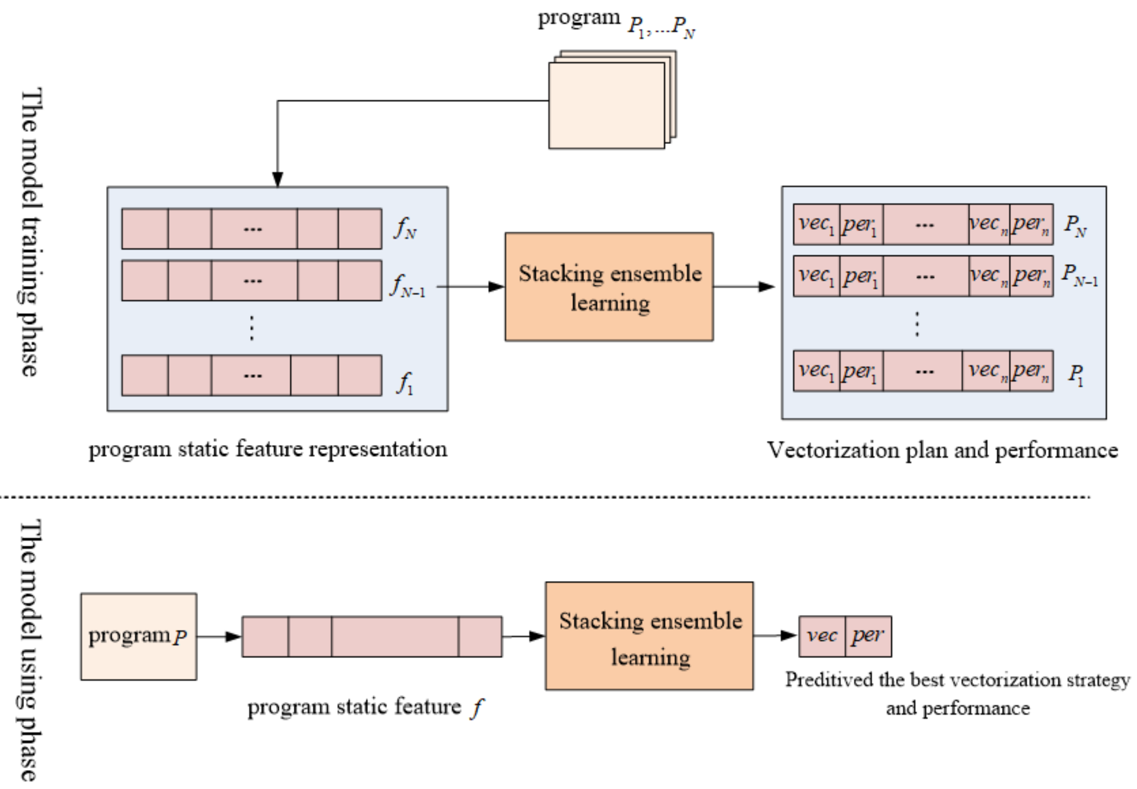

The model training phase
program $P_1, \ldots P_N$
f_N
f_{N-1}
f_1
program static feature representation
Stacking ensemble learning
vec_1 per_1 ... vec_n per_n P_N
vec_1 per_1 ... vec_n per_n P_{N-1}
vec_1 per_1 ... vec_n per_n P_1
Vectorization plan and performance
The model using phase
program P
program static feature f
Stacking ensemble learning
vec per
Preditived the best vectorization strategy and performance

FIGURE 3. Framework of automatic vectorization performance improvement model based on ensemble learning.

in prediction model. At this point, the vectorization option corresponding to the optimal vectorization performance is the optimal vectorization option we can select.

A. ENSEMBLE REGRESSION LEARNING MODEL

The ensemble learning we focus is mainly used to solve the regression problem, that is, how to find the laws from the historical samples data, establish the mapping between the input variable (program features) and the output variable (program running time). As the value of the input variable changes, the corresponding value of the output variable is obtained. The regression problem is equivalent to function fitting problem, that is, according to the known sample data set to select a function. This function can approximate the known sample data as accurate as possible, and then the new sample data can be well predicted by the function [12].

Ensemble learning is a machine learning process. Homogeneous or heterogeneous learning algorithms are used to learn the same problem in the process of ensemble learning, then multiple learners are obtained. The final results are received through a combination of the results of original learners. The ensemble learning algorithms can be divided into three types: Boosting, Bagging and Stacking. The biggest difference between these three algorithms is that the Stacking method does not simply combine the same type of learning algorithm, but uses different machine learning algorithms to construct the learning model, which may lead to a higher prediction accuracy. So we use Stacking ensemble learning technique to construct vectorization performance prediction model for TC kernels. Stacking model mainly uses a high-level learner combined with several low-level learners to achieve better predictive accuracy. Given data set $D(x, y) = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, x represents program features, y represents the performance of the vectorized program. Fig.4 is the framework of our vectorization performance prediction model based on Stacking ensemble regression learning. The pseudo code shown in Algorithm 1 corresponds to the model training phase.

First of all, we need to preprocess the data set $D(x, y)$, including data cleaning and removing imbalanced data. Since the data collected from different data sources will be integrated into data sets, there may exist data redundancy, data missing or wrong data. The introduction of these data may lead to the inaccurate prediction model, so it is necessary to clean the data set firstly. In addition, data imbalance may exists in the process of sample collection. If the model is based on this imbalanced sample set, it will tend to the majority while ignoring the minority. Therefore, we adopt the resampling strategy with synthetic new samples to optimize the imbalanced data sets, and then study on this uniform distribution sample sets.

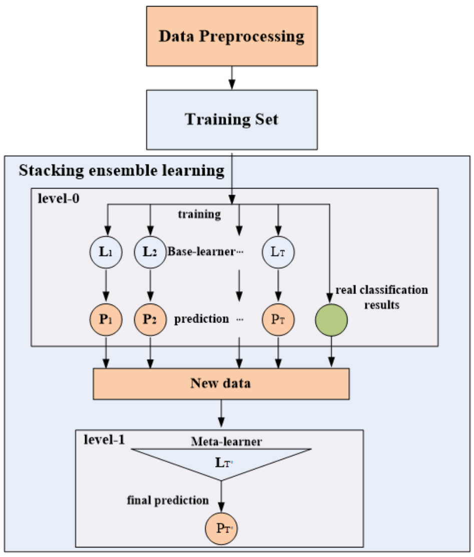


FIGURE 4. Framework of Stacking ensemble regression learning model.

Algorithm 1 Ensemble Regression Learning Algorithm

Input: Data set $D = \{(x_i, y_i)\}_{i=1}^{N}, x_i \in X \subseteq R^d, y_i \in R$, Level-0 base regression learning algorithm $\{F_1, F_2, \ldots, F_T\}$, level-1 meta regression learning algorithm $F_{T'}$.
Output: ensemble regression learning learner $L_{T'}$.

```
% Train level-0 learner $L_t$ by applying the base learning algorithms $F_T$ to the original data set $D$
1:  for t=1,2,...,T do
2:      $L_t = F_t(D)$
3:  end for
    % Generate a new data set
4:  $D_{NEW} = \varnothing$
5:  for i=1,2,...,n do
6:      for t=1,2,...,T do
            % Use $L_t$ to learn the training example $x_i$
7:          $Z_{it} = L_t(x_i)$
8:      end for
9:      $D_{NEW} = D \cup \{((Z_{i1}, Z_{i2}, \ldots, Z_{iT}), y_i)\}$
10: end for
    % train the level-1 learner by applying the learning algorithm $F_{T'}$ to the new data set $D_{NEW}$
11: $L_{t'} = F_{T'}(D_{NEW})$
```

The main structure of the framework consists of 2 layers, level-0 is the base learner, and level-1 is the meta learner. The base learner is constructed by the training set in bootstrap sampling. Multiple base learners $\{L_1, L_2, \ldots, L_T\}$ are generated by calling different kinds of learning algorithms $\{F_1, F_2, \ldots, F_T\}$ on the same training set. The learning algorithms we use include logistic regression, decision tree and linear regression. Although the sample size is large, metrics number is big, it's still be able to quickly get results at this time. Then, the predicted results of the base learners $\{P_1, P_2, \ldots, P_T\}$ are used as the input of the meta learner $L_{T'}$. In addition, the training data are also used as input to the meta learner $L_{T'}$. The predicted results and the real classification results are integrated as a new data set, then the new data set is used as the training data set of the meta learner $L_{T'}$. $L_{T'}$ is based on the new learning algorithm $F_{T'}$, and we use artificial neural network. The task of the meta learner is to reasonably combine the output set, and correct the prediction error of the base learners, then predict the target correctly. For the new TC kernel, the vectorization performance prediction by base learners are used as the inputs of the meta learner, and the final prediction result is given by the meta learner.

B. PROGRAM FEATURES

Based on the vectorization profitable assessment models in previous researches and the typical characteristics of the TC kernels, we design a new features set *VECfeatures* for the TC kernels. *VECfeatures* includes locality, dependency, whether there exists reduction operation, and memory continuity.

1) LOCALITY

The locality principle is when CPU accesses the memory, either access instructions or access data, the storage unit accessed tends to aggregate in a smaller contiguous region [14]. Locality is a predictable behavior occurring in a computer system. There are usually two representations: temporal locality and spatial locality. Temporal locality refers to the reuse of specific data and/or resources in a relatively small duration. Spatial locality refers to use data elements in relatively close storage locations. Due to the existence of loop structures, locality tends to access arrays or other data structures through indexes. When the data layout has a good locality, it can make full use of pipeline and cache to speed up memory access, and reduce the access operations of vectorization data. At the same time, the cache hit rate and access efficiency of vectorization data can also be improved.

2) DEPENDENCY

If there is such an interconnection relationship between programs codes, that is, codes A must run before codes B to ensure the normal operation of B. At this time, the interconnection is called dependency, and B depends on A [15]. The factors that influence the vectorization of SIMD are mainly due to the dependent ring formed between the statements in the loops or by the dependent edges of the statements themselves. Therefore, when analyzing the vectorization dependency relations for the loop, we can divide them into constitute a dependent ring or not.

3) REDUCTION OPERATION

The access of some arrays in TC kernels relative to the innermost loop index is discontinuous, while for the outer loop index the access is continuous. If we vectorize the innermost loop when the array access is discontinuous relative to the innermost loop index, additional vector instructions are needed for data reorganization, which will result in additional overheads. When the inner loop vectorization is going to process the reduction operations, it is necessary to insert additional reduction statements at the end of the loop and convert the vector into scalar. Reduction statements often involve complex hardware operations, and the cost is usually high. However, it's not necessary to perform such operations at the end of the loops when direct vectorizing the outer loop.

4) CONTINUITY

Memory access continuity is one of the most important factors affecting the vectorization efficiency [13]. When the data store address accessed has continuity, the data can be read into a vector register or write into memory at once, which will result in efficient vectorization access, otherwise, these data will be loaded and stored multiple times. When data access has discrete accessibility, we need to implement additional access and data operation, and reorganize target data. This will create additional overheads and reduce the vectorization income. Some CPU processors provide hardware support for non-contiguous memory access, such as the Intel MIC processor has the corresponding hardware support for gather/scatter operation. However, most processors provide data arrangement instructions, such as Sunway TaihuLight high performance computers. After reading multiple times of the required data from the memory, the compiler will then shuffle the data into the required vector units by using the arrangement instructions, and this will decrease the overall vectorization income.

For the TC kernel in Fig.1, Fig.5 shows the features set *VECfeatures* and the values. Fig.5(a) takes array B as an example to illustrate the calculation of feature values. Figure 5(b) represents the feature values of array A, B and C. *loc*(*arry*), *dep*(*arry*), *redu*(*arry*) and *con*(*arry*) represent the locality, dependency constraints, reduction operations affect the outer vectorization and continuity, respectively. *dep*(*arry*) is represented by vector form, *loc*(*arry*), *redu*(*arry*) and *con*(*arry*) are represented by matrix form.

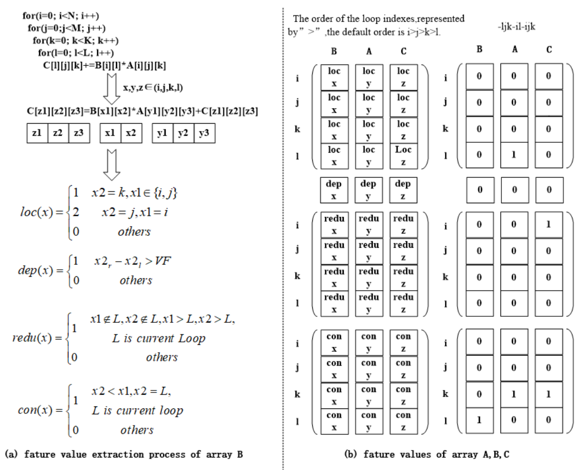

for(i=0; i<N; i++)
for(j=0;j<M; j++)
for(k=0; k<K; k++)
for(l=0; l<L; l++)
C[l][j][k]+=B[i][l]*A[i][j][k]
x,y,z∈(i,j,k,l)
C[z1][z2][z3]=B[x1][x2]*A[y1][y2][y3]+C[z1][z2][z3]
z1 z2 z3 x1 x2 y1 y2 y3
loc(x) = 1 x2 = k, x1 ∈ {i, j}; 2 x2 = j, x1 = i; 0 others
dep(x) = 1 x2_r − x2_l > VF; 0 others
redu(x) = 1 x1 ∉ L, x2 ∉ L, x1 > L, x2 > L, L is current Loop; 0 others
con(x) = 1 x2 < x1, x2 = L, L is current loop; 0 others
(a) fature value extraction process of array B
The order of the loop indexes,represented by" >" ,the default order is i>j>k>l.
-ljk-il-ijk
B A C
i j k l
loc x loc y loc z
dep x dep y dep z
redu x redu y redu z
con x con y con z
(b) fature values of array A, B, C

FIGURE 5. TC kernel features and values.

The main operation statement of the TC kernel is: $C[z1][z2][z3] = B[x1][x2] * A[y1][y2][y3] + C[z1][z2][z3]$.

The objects of feature analysis are arrays A, B, and C. Through the front-end analysis of the compiler, we can get the relevant information of the array index, and obtain the corresponding feature values. In Fig.5(a), $x1$ represents the high dimensional index of array B, $x2$ is the lowest dimensional index of array B. In array B, $x1 = i$ and $x2 = l$. When the values of $x1$ and $x2$ remain constant relative to the current loop index, array B has locality for the current loop. When the value of $x2$ increases/decreases with the current loop index, and the value of $x1$ remains the same, array B is considered to have continuity with the current loop. For the l-layer loop, the value of $x2$ increments along with the change of index l, and the value of $x1$ remains the same. For the i-layer loop, the value of $x1$ changes when index i changes. So $loc(B) = \{0, 0, 0, 0\}$ and $con(B) = \{0, 0, 0, 1\}$. Array B is read-only to the loop layer of i, j, k, l, and there is no data dependency and reduction, so $dep(B) = 0$ and $redu(B) = \{0, 0, 0, 0\}$. When the array dimension is more than 2, we take $x_1, x_2, \ldots, x_n$ (n is array dimension) to express. When the values of $x_1, x_2, \ldots, x_n$ remain constant relative to the current loop index, array B has locality for the current loop. When the value of x_n increases/decreases with the current loop index, and the values of $x_1, x_2, \ldots, x_{n-1}$ remain the same, array B is considered to have continuity with the current loop.

C. OPTIMIZE SPACE

The input of our ELAV model is *VECfeatures* of TC kernel, the output is the running time prediction of the kernel when using different vectorization strategies. Based on the *VECfeatures*, ELAV model can predict the best program performance by vectorizing the suitable loop layer. Furthermore, in order to fully use instructions pipeline and cache pipelining parallel, we also predict the optimal unrolling factor for the vectorized loop layer. The vectorization strategy is shown in Table.1, and the command line is shown in Table.2. Table.1 indicates that we can choose which layer (our TC kernels is up to 4 layers) to execute vectorization, or do not vectorize any loop layer. When a loop layer is vectorized, its loop unroll factor can be specified as $\{1, 2, 4, 8, 16\}$.

TABLE 2. Units for magnetic properties.

		SSE	AVX
Default optimization	ICC	icc -xSSE -O3	icc -xAVX -O3
	GCC	gcc -msse4.2 -O3	gcc -mavx -O3
Force vectorization	ICC/GCC	#pragma ivdep	#pragma ivdep
unroll factor	ICC	icc -xSSE -unroll(n)	icc -xAVX -unroll(n)
	GCC	gcc -msse -unroll(n)	gcc -mavx -unroll(n)

IV. EXPERIMENTAL AND RESULTS

Platform I: Intel core processor Ivy Bridge, CPU is Xeon E5-2697A v4 2.6GHz, memory is 128G, and the width of vector register is 256 bits.

Platform II: Intel Xeon Phi Xeon processor Knights Landing fusion (KNL), CPU is Intel Xeon Phi Processor 7210, and the width of vector register is 512 bits.

We compose Ivy Bridge and KNL platforms, SSE and AVX instruction sets, float and double data types, Intel ICC 12.0 and GCC 4.6 compilers into 16 different configurations shown in Table.3. We randomly select 1100 TC kernels from CCSD for model training, and set the array dimension N to 10-200, so we have 1100 kernels$\times$191 sizes = 210100 data points. The input of ELAV model is the TC features *VECfeatures*, and the output is the predicted vectorization performance when using different vectorization strategies. During model testing period, we predict the vectorization performance of 288 new TC kernels (different with training sets) in CCSD to verify the accuracy and efficiency of the model.

TABLE 3. Configurations based on different platforms, instruction sets, data types and compilers.

IADG	IADI	IAFG	IAFI	ISDG	ISDI	ISFG	ISFI
KADG	KADI	KAFG	KAFI	KSDG	KSDI	KSFG	KSFI

The experiment is carried out from three aspects: Firstly, we analyze the vectorization performance of TC kernels by different instruction sets, data types and compilers on Ivy Bridge and KNL platforms, and compare the performance differences when using the default vectorization rules and the forced vectorization for TC kernels. Secondly, through taking random features, VECfeatures, Milepost features and assembly features as model input, we compare the model prediction accuracy for the TC kernels in the test set. Finally, we compare the prediction efficiency when using different predictive models, and the prediction efficiency of different models with different configurations.

A. VECTORIZATION PERFORMANCE

Our goal is to build the ELAV model to choose the best vectorization strategy for different TC kernels, in order to improve the vectorization performance of the kernels. We first analyze the vectorization performance on different platforms. The degree of performance improvement is expressed as:

$$Speedup_{vec} = \frac{T_{default}}{T_{vec}}$$

In this formula, $T_{default}$ represents program runtime when using the default vectorization rules of the compiler, T_{vec} represents the runtime of forced vectorization. We divide the $Speedup_{vec}$ into class1: $Speedup_{vec} > 1$, class2: $0.5 \leqslant Speedup_{vec} \leqslant 1$ and class3: $Speedup_{vec} < 0.5$. As the experimental results are numerous, we use IADG and KSFI shown in Fig.6 and Fig.7 as examples to illustrate. The horizontal axis represents the number of data points sorted in ascending performance, and the vertical axis represents the vectorization speedup.

The gray lines in Fig.6 and Fig.7 correspond to the default behavior of the compiler, and the black lines represent the forced vectorization. When the black line is higher than the gray, it means that the compiler is lack of the opportunity to vectorize the TC kernels, and the forced vectorization is

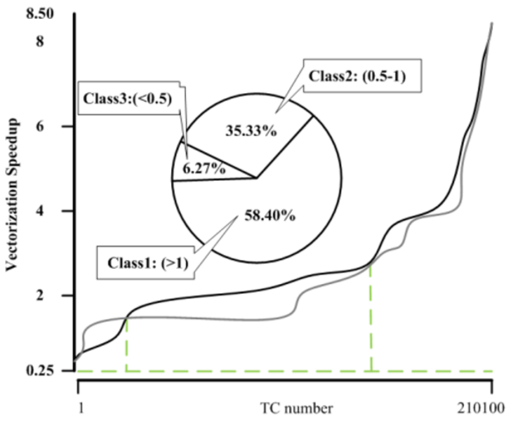


FIGURE 6. Vectorization performance analysis on Ivy Bridge platform, using the AVX instruction set, double data type and GCC 4.6 compiler.

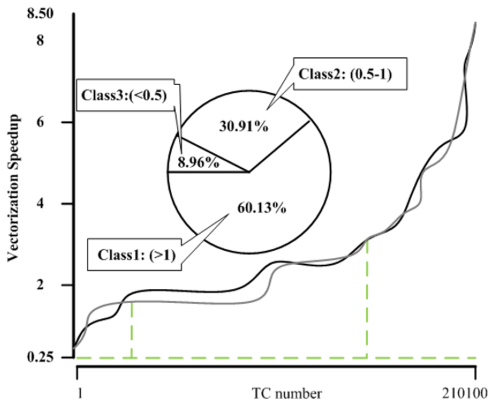


FIGURE 7. Vectorization performance analysis on KNL platform, using the SSE instruction set, float data type and ICC 12.0 compiler.

beneficial to improve program performance. At this time, $Speedup_{vec}$ belongs to class1.When the gray line is higher than the black line, it means that the compiler default vectorization behavior is the good, and $Speedup_{vec}$ belongs to class2. When the gray line is much higher than the black line, it indicates that the forced vectorization will result in a more longer compilation time, and $Speedup_{vec}$ belongs to class3.

We can see from Fig.6, the GCC compiler can make the correct vectorization decision in 35.33% of the data points. For 58.40% of the data, the default vectorization rule is too conservative. Because these programs are not vectorized, average 32$\times$speedup of the program execution time is lost. In addition, due to force vectorization, 6.27% of the data results in a longer runtime. From Fig.7 we can see the GCC compiler can make the correct decision in 30.91% of the points. The default vectorization rule is too conservative in 60.13% of the points. 8.96% of the data results in a longer runtime. Therefore, in order to improve the automatic vectorization performance of the programs effectively, we need to determine the optimal vectorization strategies for different TC kernels.

B. FEATURES REPRESENTATION

1) PREDICTION ACCURACY WITH DIFFERENT FEATURES

In order to illustrate the prediction accuracy of ELAV model when using different program features, we compare 4 different kinds of features set: (*i*) Random features, (*ii*) *VECfeatures*, (*iii*) Milepost static features, (*iv*) Assembly features. Among them, four random static and dynamic features et al. [16] are selected as the baseline. If the features set we choose produces a precision similar to the random features, it means that these features have poor program representations. The Milepost static features we selected are the same as Fursin *et al.* [17]. There are 56 static features, including number of basic blocks in the function, number of edges in the control flow graph and so on. Assembly features are selected from Stock *et al.* [18], including vector operation counts, arithmetic intensity, sufficient distance, sufficient distance ratio, total operations, and critical path. The prediction accuracy of IADG is shown in Fig.8. Class1, class2, and class3 represent $Speedup_{vec}$ classification in section 4.1.

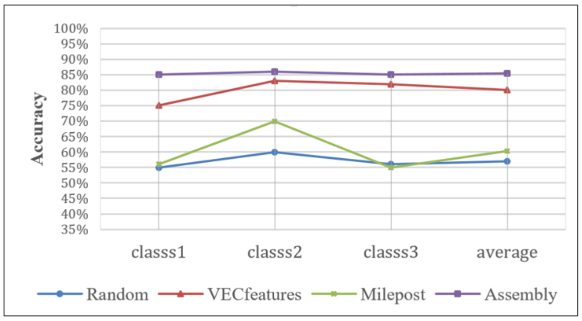


FIGURE 8. Prediction accuracy of ELAV when using different features selection methods.

As shown in Fig.8, when the prediction model uses the program features generated by Random, the average prediction accuracy is low, and each class is around 50%. The prediction accuracy of Milepost features is slightly higher than the Random, but the accuracy differences between each class is relatively large. Assembly features can provide higher prediction accuracy, because the vectorization performance benefits mainly come from vectorized instructions which automatic generated by the compiler, such as load/store and arithmetic operations. The prediction accuracy generated by *VECfeatures* is obviously higher than Random and Milepost features. The accuracy generated of *VECfeatures* is 80%, which is slightly lower than Assembly features. However, *VECfeatures* is extracted before the compiling, so the features extraction process does not require to actually compile the program, while the assembly features are extracted after the compiling.

2) PREDICTION PERFORMANCE WITH DIFFERENT FEATURES

When the ELAV model is used to predict the vectorization performance for the new TC kernels, if the prediction is in class1, we use the command line of the compiler to force the vectorization. If the prediction is in class2 or class3, the compiler default optimization is used. We choose IADG configuration for an example. The $Speedup_{vec}$ distribution of the test set TC kernels is shown in Fig.9, which shows the average speedup of the program performance obtained by using different feature sets. Among them, manual is the result of manual vectorization, which represents the maximum speedup.

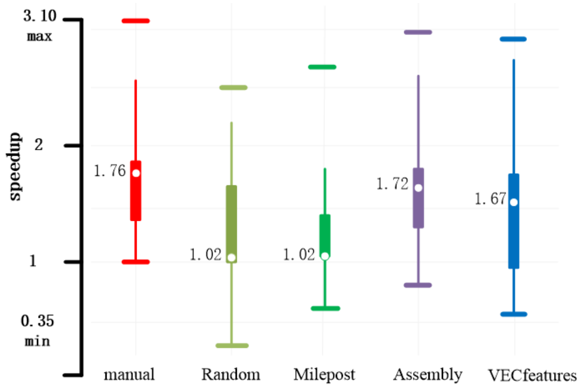


FIGURE 9. Speedup distributions using different features representation.

As shown in Fig.9, manual vectorization achieves 1.76× average speedup. By using the *VECfeatures* feature set, we receive 1.67× average speedup, which is about 95% of the optimal. When using Assembly features, we can achieve 1.72× average speedup. However, the Assembly features need to be obtained after the program compiling. The Random and Milepost features both have 1.02× average speedup, which are relatively lower. Therefore, *VECfeatures* can better represents the TC kernels, improve the performance of the prediction model and reduce the training time cost. The reason about *VECfeatures* sometimes make a performance prediction bias is when choosing a loop layer for vectorizaion, sometimes may cause part of the data access is not continuous. At this time, we need to implement data reorganization and operation with some special or un-aligned vector instructions. In addition, repeated reading operations for some loop invariants may result in unnecessary access overheads, and make program vectorization performance reduction.

C. TEST SET PERFORMANCE

1) PREDICTION EFFICIENCY OF DIFFERENT LEARNING MODELS

In order to verify the vectorization prediction efficiency of the ELAV method, we compare the prediction results with the single machine learning algorithm and the WeightedRank model proposed by Stock *et al.* [18]. The single learning algorithm involves six different learning algorithms provided in Bouckaert [19]. These learning algorithms use a single learner, while the ELAV method uses Stacking ensemble learning algorithms, including logistic regression, decision tree, linear regression, and artificial neural network. WeightedRank is a model based on features extracted from the generated assembly code, which can effectively improve the vectorization performance of the programs. The prediction efficiency is calculated as follows:

$$Efficiency = \frac{Performance\ of\ predicted\ best}{Performance\ of\ actual\ best}$$

If the predicted best vectorization strategy is the actual best for a particular TC kernel, the prediction efficiency is 100%. By evaluating all possible vectorization strategies, we find the vectorization version with the greatest performance improvement and use it as the actual best.

When we use AVX instruction set, double data type and GCC 4.6 compiler on the Ivy Bridge platform, we have 288 valid test data. We take 25 for an example, the prediction efficiencies of different models are shown in Table.4. IBk and K* are learning algorithms based on instances, which implement prediction based on similar instances in the training set. LR represents linear regression. M5P represents M5 model tree algorithm that combine the tree structure and linear regression model. MLP represents the multilayer perceptron, and it's a non-loop artificial neural network using the back-propagation training. SVM means support vector regression algorithm. When we use these single learning models and our ELAV ensemble learning model, the model input is VECfeatures, and the output is vectorization performance prediction. ICC and GCC represent the efficiency of using compiler default vectorization, Rand represents the results of a random choice from the search space of vectorization variants, and WR represents the WeightedRank model. We use three array indexes to represent the TC kernel. For example, ij-ik-kj represents the contraction of two 2-dimensional tensors to produce a 2-dimensional tensor, and the loop nesting depth is 3.

TABLE 4. Prediction efficiency of different models on Ivy Bridge platform, using AVX instruction set, double data type and GCC 4.6.

TC	IBK	K*	LR	M5P	MLP	SVM	ICC	GCC	Rand	WR	ELAV
ij-ik-kj	0.80	0.98	0.76	0.88	0.85	0.67	0.34	0.36	0.38	0.93	0.99
ij-ilk-ljk	0.75	0.73	0.85	0.76	0.71	0.83	0.30	0.29	0.36	0.86	0.85
ij-kil-ljk	0.90	0.90	0.65	0.94	0.67	0.71	0.30	0.35	0.31	0.87	0.92
ijk-ikl-lj	0.87	0.76	0.61	0.93	0.83	0.65	0.34	0.32	0.44	0.89	0.88
ijk-il-jlk	0.99	0.95	0.73	0.79	0.69	0.85	0.43	0.41	0.32	0.95	0.96
ijk-lk-ijl	0.74	0.91	0.57	0.96	0.87	0.83	0.45	0.47	0.36	0.92	0.94
ijk-ilk-lj	0.82	0.89	0.67	0.85	0.78	0.88	0.44	0.42	0.41	0.81	0.90
ijk-jl-ikl	0.85	0.71	0.54	0.79	0.73	0.82	0.41	0.39	0.34	0.91	0.89
ijk-jil-kl	0.81	0.77	0.68	0.84	0.81	0.85	0.37	0.38	0.37	0.85	0.87
ijk-ilmk-mjl	0.84	0.68	0.63	0.44	0.73	0.77	0.28	0.30	0.30	0.83	0.87
ijkl-imjn-lnkm	0.91	0.83	0.77	0.79	0.86	0.61	0.36	0.31	0.35	0.95	0.95
ijkl-imjn-nlmk	0.87	0.95	0.77	0.74	0.78	0.57	0.30	0.32	0.33	0.93	0.96
ijkl-imkn-jnlm	0.92	0.93	0.72	0.68	0.66	0.57	0.35	0.34	0.28	0.96	0.98
ijkl-imnk-njml	0.89	0.96	0.94	0.84	0.89	0.75	0.29	0.29	0.31	0.91	0.93
ijkl-imln-njkm	0.80	0.92	0.78	0.83	0.69	0.70	0.45	0.41	0.27	0.90	0.91
ijkl-imln-njmk	0.88	0.86	0.52	0.92	0.76	0.49	0.33	0.29	0.28	0.88	0.87
ijkl-imnj-nlkm	0.83	0.97	0.68	0.85	0.77	0.58	0.35	0.36	0.37	0.86	0.92
ijkl-imnk-njml	0.79	0.76	0.87	0.78	0.83	0.44	0.28	0.30	0.41	0.83	0.84
ijkl-minj-nlmk	0.90	0.88	0.67	0.86	0.93	0.68	0.32	0.35	0.45	0.91	0.92
ijkl-mink-jnlm	0.76	0.93	0.47	0.91	0.82	0.50	0.29	0.30	0.34	0.83	0.86
ijkl-minl-njmk	0.94	0.95	0.67	0.93	0.73	0.58	0.39	0.34	0.42	0.95	0.97
ijkl-mikn-njml	0.71	0.83	0.80	0.82	0.79	0.67	0.31	0.32	0.31	0.79	0.87
ijkl-mlin-nmjk	0.81	0.81	0.68	0.90	0.85	0.74	0.35	0.33	0.38	0.79	0.82
ijkl-miln-njmk	0.64	0.88	0.73	0.84	0.81	0.76	0.42	0.43	0.45	0.78	0.8
ijkl-niml-jmnk	0.76	0.79	0.81	0.76	0.82	0.84	0.44	0.40	0.35	0.81	0.83
Average	0.83	0.86	0.70	0.83	0.79	0.69	0.36	0.35	0.36	0.88	**0.90**

We can see from Table.4 that the average prediction efficiency of the ELAV model is 90%. The good predictive efficiency is mainly due to our way of features extraction and the use of ensemble regression model. The second best is WeightedRank, but the use of this model requires to compile the kernels at least once. The process of our features extraction doesn't need to actually compile the kernels. The prediction efficiency of ELAV model is better than all single learning algorithms. When using ICC or GCC compiler default vectorization rule, the prediction efficiency is relatively lower. The prediction efficiency of Rand is only slightly better than ICC and GCC compilers.

From Table.4 we can see that the ELAV, WR, and IBK learning models have higher prediction efficiency. The prediction time overheads of these three models are shown in Table.5. From Table.5 we can see that our ELAV has a minimal prediction time overheads. Therefore, ELAV method has the highest prediction accuracy and the lowest prediction time overheads when comparing with existing methods.

TABLE 5. The prediction time overheads of different models(s).

		Model	Ivy Bridge platform platform	KNL platform
Offline learning	Data collection	ELAV	6.048×10^5	6.481×10^5
		WR	9.504×10^5	9.936×10^5
		IBK	11.232×10^5	11.232×10^5
	Model construction	ELAV	35.20	34.90
		WR	57.30	60.40
		IBK	55.60	55.30
Online prediction	Feature extraction	ELAV	9.30	10.30
		WR	11.30	13.20
		IBK	12.60	14.10
	Model prediction	ELAV	0.58	0.57
		WR	0.73	0.83
		IBK	0.69	0.69

2) PREDICTION EFFICIENCY OF DIFFERENT CONFIGURATIONS

Table.6 and Table.7 indicate the prediction efficiency of different learning models on Ivy Bridge and KNL platforms with different instruction sets, data types and compilers. From Table.6 we can see that the prediction efficiency of our ELAV model is above 80% in all configurations, and the average prediction efficiency is 88%, which is also higher than the

TABLE 6. Prediction efficiency of learning models on Ivy Bridge platform, using different instruction sets, data types and compilers.

Model	IADG	IADI	IAFG	IAFI	ISDG	ISDI	ISFG	ISFI	Average
IBk	83%	84%	76%	82%	85%	78%	75%	86%	81%
K*	86%	85%	75%	83%	79%	86%	81%	73%	81%
LR	70%	67%	71%	78%	86%	82%	65%	73%	74%
M5P	83%	75%	80%	57%	52%	63%	77%	83%	71%
MLP	79%	75%	68%	70%	55%	63%	52%	58%	65%
SVM	69%	78%	67%	63%	72%	80%	83%	58%	71%
Random	36%	49%	45%	54%	47%	41%	46%	39%	45%
WR	88%	86%	73%	85%	83%	86%	90%	89%	85%
ELAV	**90%**	87%	89%	88%	82%	89%	89%	**90%**	**88%**

TABLE 7. Prediction efficiency of learning models on KNL platform, using different instruction sets, data types and compilers.

Model	KADG	KADI	KAFG	KAFI	KSDG	KSDI	KSFG	KSFI	Average
IBk	80%	85%	78%	82%	89%	79%	86%	88%	83%
K*	83%	87%	79%	80%	85%	89%	81%	76%	83%
LR	57%	72%	82%	75%	86%	70%	64%	72%	72%
M5P	64%	73%	82%	53%	49%	68%	75%	83%	68%
MLP	68%	72%	77%	62%	57%	66%	54%	63%	65%
SVM	72%	80%	65%	59%	69%	77%	85%	53%	70%
Random	52%	47%	44%	50%	41%	39%	46%	40%	45%
WR	83%	82%	75%	83%	83%	84%	87%	88%	83%
ELAV	85%	86%	81%	85%	90%	91%	**93%**	89%	**87%**

other models. The ELAV model can achieve average 90% prediction efficiency under IADG and ISFI configurations. The average prediction efficiency of WR model is 85%, which is close to our method, but WR is implemented after the compiling. Therefore, our method is better than WR both in prediction accuracy and prediction time. Random method has the lowest prediction efficiency. From Table.7 we can see that the prediction efficiency of ELAV model is above 80% in all configurations, and the average prediction efficiency is 87%, which is also higher than the other models. The ELAV model can achieve average 93% prediction efficiency under KSFG configurations. ELAV model analyzes the dependence and interaction between program transformations, and considers the vectorization along all dimensions, rather than the data access with unit step. In terms of the overall testing results, the ELAV model achieves the best prediction efficiency on both Ivy Bridge and KNL platforms.

In addition, we use the general floating point performance indicator GFlop/s to measure the performance of TC kernels. Table.8 shows the performance obtained by different compilers and ELAV model under different configurations. Min and Max represent the lowest and highest TC performance, and AVG means the average performance. ELAV/ICC and ELAV/GCC shows the average performance ratio of ELAV to ICC and GCC, respectively.

TABLE 8. Comparison of program performance between compiler automatic vectorization and ELAV model under different configurations.

configuration	ICC			GCC			ELAV			ELAV/ICC	ELAV/GCC
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Rate	Rate
IADG	1.25	2.76	5.96	1.32	2.73	5.18	7.56	12.78	**15.85**	4.63	4.68
IADI	1.43	4.97	14.70	1.28	4.34	13.90	9.65	12.07	15.24	2.43	2.78
IAFG	1.64	3.52	6.24	1.30	2.82	5.59	6.63	7.64	8.57	2.17	2.71
IAFI	1.38	3.82	8.98	1.27	3.12	8.27	3.45	5.87	7.96	1.54	1.88
ISDG	2.05	4.17	7.63	2.83	5.26	9.01	5.67	9.82	11.28	2.35	1.87
ISDI	1.78	3.97	9.02	1.82	3.90	8.86	4.71	6.29	8.85	1.58	1.61
ISFG	2.78	3.56	6.85	2.85	3.66	7.10	6.87	8.99	9.94	2.53	2.46
ISFI	1.55	4.05	8.32	1.58	4.11	8.79	8.38	10.27	14.62	2.54	2.50
KADG	1.96	4.28	9.38	1.78	4.16	8.58	8.96	15.28	18.79	3.57	3.67
KADI	2.05	4.28	10.93	1.86	4.15	9.76	24.74	25.69	**31.30**	6.00	6.19
KAFG	1.82	3.73	7.65	1.89	3.90	6.69	9.97	10.86	12.71	2.91	2.78
KAFI	2.58	4.57	12.81	2.16	4.38	10.45	6.47	8.94	10.38	1.96	2.04
KSDG	2.27	6.13	18.28	2.77	6.69	19.14	10.12	16.65	20.21	2.72	2.49
KSDI	2.73	7.07	21.03	2.95	7.77	21.47	14.27	23.79	28.48	3.36	3.06
KSFG	1.82	3.42	7.93	1.92	3.62	7.86	11.72	14.36	21.23	4.20	3.97
KSFI	2.68	4.75	13.68	2.36	4.53	11.57	10.47	13.75	16.68	2.89	3.04
Average	1.99	4.32	10.59	2.01	4.32	10.14	9.35	12.69	15.76	**2.96**	**2.98**

We can see from Table.8 that compared with the automatic vectorization of ICC and GCC, the ELAV model can achieve larger floating-point performance gains. For the Ivy Bridge platform, the floating point performance reaches the highest 15.85GFlop/s under IADG configuration. For the KNL platform, the highest floating-point performance is achieved by KADI configuration, with the value of 31.3GFlop/s. In addition, the data in Table.8 also shows that for some configurations, the performance gains obtained by GCC compiler is better than ICC compiler. The reason is some optimization strategies of GCC compiler is more effectively than ICC, such as loop unroll-and-jam.

V. RELATED WORK

Compiler optimization methods based on machine learning implement compiler optimization prediction by artificial intelligence means, through convert the target problem into the forms which the machine learning model can accept, the methods can search the best optimization from the optimization space more effectively. Previous researches utilize the Nearest Neighbor algorithm (NN) [2], Support Vector Machine (SVM) [17], Artificial Neural Network (ANN) [20], Logistic Regression (LR) [18] and other machine learning techniques to improve the compiler optimization.

In order to find the optimal optimization order for the particular programs, Agakov *et al.* [2] bias the existing random or genetic search algorithms via NN, and reduce the optimization search time by an order of magnitude. Pekhimenko and Brown [23] use logistic regression to determine the optimal parameters of program transformations, and they compare the execution time of the program when using the optimization parameters determined by logistic regression and compiler default. Ding *et al.* [24] propose a variable input analysis method, they construct model to automatically determine what algorithms should be used to implement optimize when different optimization strategies suitable for different inputs. Martins *et al.* [25] propose an application related optimization selection method by clustering design space explore technology guided by genetic algorithm.

Program feature representation technology can be divided into static feature representation and dynamic feature representation. Fursin *et al.* [17] develop GCC milepost, including 46 static features extracted from the intermediate representation of GCC. Park *et al.* [22] propose a different method to define the static features, and use graph mining technology to supply the program data stream graph to SVM. Hoste and Eeckhout [26] use the dynamic features to obtain the program information. However, all of these feature extraction methods have a common drawback: they only consider the properties of the control flow graph, or the static counts for each type of instructions. But the TC kernels we consider have the same control flow graph and instruction sequence, so all of these methods are not applicable.

The work of Stephenson and Amarasinghe [21] and Stock *et al.* [18] is closest to our work. Stephenson uses multiply classification algorithm to predict the loop unrolling factor that can produce the best performance for the programs. They use SVM and NN to predict the best unrolling factor for the new programs. The prediction accuracy is 65% and 62%, respectively. Our work differs from Stephenson in that the vectorization strategy optimization space we considered is not limited to the loop unrolling factor. Stock's work is a superset of our work, they consider a larger optimization space. However, our work has two different characteristics. One is that we propose a very different static features set for TC kernels. The second is that we extract features before the actual optimization, while Stock's requires to actually compile the program at least once to extract the features.

Our work aims to predict the vectorization strategies for TC kernels through machine learning model. The vectorization strategies include whether or not turn on the force vectorization option of the compiler, which loop layer should be vectorized and the value of the loop unrolling factor. To the best of our knowledge, our research content is first proposed, which is a useful complement to the relevant work. The biggest differences between our work and the existing lie in two points: (*i*) we use ensemble learning method to construct machine learning model, and ensemble learning has better prediction accuracy than a single machine learning algorithm. (*ii*) we propose a new program features representation, which is more suitable for TC kernels.

VI. CONCLUSION

Most commercial compilers can perform automatic vectorization for multi-core processors with short vector SIMD instruction set, but the performance obtained is usually far below the peak performance of the processor. Compiler automatic vectorization heuristics are often too conservative, and sometimes lose a lot of vectorization opportunities. We propose an automatic vectorization performance improvement method ELAV. The input of ELAV model is *VECfeatures* of TC kernels, which is a static feature representation method to represent storage access patterns of TC kernels. The output of ELAV model is the program performance prediction when using different vectorization strategies. The average prediction time of the ELAV model only takes a few seconds, so the model is easier to embed into the existing compilation flow. With different instruction sets, data types and compilers, the prediction efficiency of ELAV model on Ivy Bridge and KNL platforms is 88% and 87%, respectively. At the same time, the prediction efficiency of ELAV model is higher than that of other single learning algorithm on two platforms. In addition, the average peak performance obtained by ELAV model under different configurations is 2.96× of Intel ICC 12.0 and 2.98× of GCC 4.6 compiler, respectively.

We believe that the ELAV method still has some follow-up work worth researching, specifically: (*i*) we will consider more test sets and target platforms to further improve the prediction accuracy and generality of the ELAV method. (*ii*) we will research more relevant features of different optimizations to further improve the performance of TC programs. (*iii*) we will try to increase the robustness of the ELAV method by adding artificial noise, and investigate the performance of the ELAV method in the heavily loaded multi-user environment.

REFERENCES

[1] T. D. Crawford and H. F. Schaefer, III, "An introduction to coupled cluster theory for computational chemists," *Rev. Comput. Chem.*, vol. 14, pp. 33–136, Jan. 2007.

[2] F. Agakov *et al.*, "Using machine learning to focus iterative optimization," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, Mar. 2006, p. 305.

[3] E. Park, S. Kulkarni, and J. Cavazos, "An evaluation of different modeling techniques for iterative compilation," in *Proc. Compil., Archit. Synth. Embedded Syst. (CASES)*, 2011, pp. 65–74.

[4] A. Monsifrot, F. Bodin, and R. Quiniou, "A machine learning approach to automatic production of compiler heuristics," in *Proc. Artif. Intell., Methodol., Syst., Appl. (AIMSA)*, 2002, pp. 41–50.

[5] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2010, pp. 190–199.

[6] A. Chandra and X. Yao, "Ensemble learning using multi-objective evolutionary algorithms," *J. Math. Model. Algorithms*, vol. 5, no. 4, pp. 417–445, 2006.

[7] A. Trouvé *et al.*, "Using machine learning in order to improve automatic SIMD instruction generation," *Procedia Comput. Sci.*, vol. 18, pp. 1292–1301, Jan. 2013.

[8] R. J. Harrison *et al.*, "Multiresolution computational chemistry," *J. Phys., Conf.*, vol. 16, no. 1, p. 243, 2005.

[9] J. Cavazos and M. F. P. O'Boyle, "Method-specific dynamic compilation using logistic regression," in *Proc. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2006, pp. 229–240.

[10] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," in *Proc. Program. Lang. Des. Implement. (PLDI)*, 2003, pp. 77–90.

[11] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Proc. Int. Symp. Code Gener. Optim.*, 2007, pp. 185–197.

[12] R. A. Berk, *Statistical Learning from a Regression Perspective* (Springer Series in Statistics). New York, NY, USA: Springer, 2008.

[13] Y. Yao and R. C. Zhao, "Continuity analysis algorithm based on profile information and its optimization," *Comput. Eng.*, vol. 38, no. 9, pp. 28–31, 2012.

[14] J. Shin, J. Chame, and M. Hall, "Exploiting superword-level locality in multimedia extension architectures," *J. Instruct. Level Parallelism*, vol. 5, pp. 1–28, Apr. 2003.

[15] J. Zhao *et al.*, "An improved nonlinear data dependence test," *J. Supercomput.*, vol. 71, no. 1, pp. 340–368, 2015.

[16] H. Liu, R. C. Zhao, and Q. Wang, "Function level compiler optimization parameters selection method based on supervised learning model," in *Proc. Nat. Annu. Conf. High Perform. Comput. (HPC)*, 2017, pp. 168–176.

[17] G. Fursin *et al.*, "Milepost GCC: Machine learning enabled self-tuning compiler," *Int. J. Parallel Program.*, vol. 39, no. 3, pp. 296–327, 2011.

[18] K. Stock, L. Pouchet, and P. Sadayappan, "Using machine learning to improve automatic vectorization," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–23, Jan. 2012.

[19] R. R. Bouckaert *et al.*, "WEKA—Experiences with a java open-source project," *J. Mach. Learn. Res.*, vol. 11, no. 5, pp. 2533–2541, 2011.

[20] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proc. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2012, pp. 147–162.

[21] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *Proc. Int. Symp. Codegeneration Optim.*, Mar. 2005, pp. 123–134.

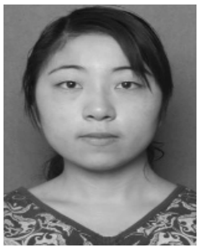
[22] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proc. Int. Symp. Code Gener. Optim. (CGO)*, 2012, pp. 196–206.

[23] G. Pekhimenko and A. D. Brown, *Efficient Program Compilation Through Machine Learning Techniques* (Software Automatic Tuning). New York, NY, USA: Springer, 2010.

[24] Y. F. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proc. Program. Lang. Des. Implement. (PLDI)*, pp. 379–390, 2015.

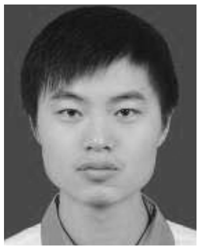
[25] L. G. A. Martins *et al.*, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, 2016, Art. no. 8.

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



HUI LIU received the master's degree from the College of Computer and Information Engineering, Henan Normal University, Xinxiang, China, in 2008. She is currently pursuing the Ph.D. degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Information Engineering University, Zhengzhou, China. She has been with the College of Computer and Information Engineering, Henan Normal University, since 2008, where she is also a Lecturer. Her research interests include high-performance computing, optimizing compilers, and machine learning.

RONGCAI ZHAO received the Ph.D. degree from the Chinese Academy of Sciences, Beijing, China, in 2002. He is currently a Professor with the State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Information Engineering University, Zhengzhou, China, and also a Professor with the Zhongyuan University of Technology, Zhengzhou. His research interests include high-performance computing, program performance analysis, and machine learning. He is a member of the ACM and the Henan Computer Federation.



KAI NIE (M'87) received the master's degree from the School of Information Engineering, Zhengzhou University, Zhengzhou, Henan, China, in 2017. He is currently pursuing the Ph.D. degree with the State Key Laboratory of Mathematical Engineering and Advanced Computing, PLA Information Engineering University, Zhengzhou. His research interests include high-performance computing, optimizing compilers, and machine learning.