Sustainable Learning-Based Optimization Based on RKNN Outlier Detection

Shun Long

[1] Department of Computer Science, JiNan University, Guangzhou 510632, P.R.China
[2] Key Laboratory of Computer System and Architecture, Institute of Computing Technology, China Academy of Science, Beijing 100080, P.R.China
tlongshun@jnu.edu.cn

Abstract. Iterative compilation has been proved a successful approach to achieve high performance, particularly if enhanced with machine learning techniques. However, we point out in this paper that the capability of such learning-based compilers relies heavily on the training examples chosen, which hinders their applicability in general scenarios. To tackle this pitfall, we use reverse K-nearest neighbor (RKNN) algorithm to help a compiler to decide whether to use existing prior experience directly, or turn to launch an optimization space search for outlier programs instead. This approach is proposed as a supplement instead of a replacement of the existing learning-based iterative optimizations, in order to make the latter sustainable and capable of dealing with arbitrary programs given. Preliminary experimental results are given to demonstrate its effectiveness.

Keywords: iterative optimization; machine-learning; program feature; outlier; Reverse K-Nearest Neighbours

1 Introduction

Modern compilers use various optimization heuristics and performance models in order to fully exploit the potential of underlying hardware in pursuit of higher performance. However, fixed heuristics and static performance models have difficulties in coping with the rapidly evolving nature of modern architectures, and therefore have difficulties in providing portable high performance. Compilers are expected to evolve accordingly, i.e. capable of automatically fine-tuning itself for a given architecture, without dramatic change to its internal structure.

Iterative optimization[6] is a promising approach to achieve portable high performance. Given a program, an iterative compiler explores an optimization space composed of various transformations, in search for good points (i.e. transformation sequences) which yield high performance. Various approaches[1,4,5,15] have been proposed to apply machine learning[14] techniques to accelerate optimization space exploration. They can not only effectively identify good points within the space for a given program, but also accelerate the exploration process via the use of the compiler's prior optimization experience with various programs.

Funded by 1) The Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, 2) State Key Laboratory of Software Engineering in Wuhan University (SKLSE2010-08-31) and 3) The Science and Technology Planning Project of Guangdong Province, China (No.2010A032000002).

However, most of these approaches are based on the assumption that the compiler has been properly trained with good-enough examples, i.e. programs which demonstrate significant performance improvement if proper transformations are applied. Usually, compiler developers carefully choose these training examples with hopes that they will suit the future programs. Nevertheless, when an unfamiliar new program (i.e. an outlier) is encountered, its performance remains a doubt. This indicates that training example selection plays a decisive part in the success of learning-based optimization. To the best of our knowledge, this problem has not been properly addressed yet.

To put learning-based optimization into practice, we must address this pitfall of these current approaches. We propose in this paper the idea of using reverse K-nearest neighbor (RKNN)[7] approach to achieve a sustainable iterative compiler framework whose performance evolves with arbitrary programs encountered and is therefore independent of the training examples provided. Given arbitrary programs, it can decide whether to explore from scratch the predefined optimization space composed of various transformations, or to directly apply its experiences learned from other familiar programs it has optimized before. This decision is based on its observation on the given program and those it has optimized before, particularly the similarities in between. Preliminary experimental results show that it can make proper decisions when given a sequence of programs, and its optimization effectiveness gradually improves as more experiences are learned from the training examples.

The organization of this paper is as follows. First, section II briefly reviews the learning-based optimization paradigm, particularly its pitfall in its dependency on training examples provided. Section III first presents our goal of a sustainable learning based optimization whose performance does not rely on training examples explicitly given. It then presents a brief introduction of the RKNN algorithm, before discusses the RKNN-based approach which the compiler uses to identify outlier programs. Our implementation is then given in section IV, together with some preliminary experimental results. Some related works are discussed in section V, before some concluding remarks in section VI.

2 Learning Based Optimization and Its Training Example Pitfall

A typical iterative compilation paradigm is outlined in [6]. When a program is encountered, the compiler uses some heuristics to generate an optimized version, before launching a test run. It then decides whether or not to make further attempts based on the runtime profiles collected. This process is repeated until a good enough optimization is found or the optimization budget is reached.

In search of good optimization scheme for each program encountered, the compiler has a large optimization space to explore, which is composed of arbitrary combinations of a collection of transformations, together with their corresponding options and parameters. Various algorithms have been proposed to accelerate this iterative process. They are based on either heuristic search[2,6,8,17] or machine learning techniques[1,4,15], or a hybrid solution[5]. These prior works have demonstrated the efficiency of learning based approaches.

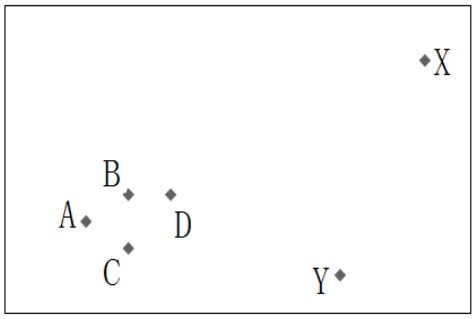

Fig.1. Given six programs within a feature space, A, B, C and D are close to each other and therefore considered similar. A compiler may directly apply its experience with anyone to the other three, and vice versa. However, X and Y are far away from them and therefore considered as outliers. The compiler cannot apply its experience with A, B, C and D to these outliers.

However, it is worth noting that the success of learning-based optimization rests on not only on the learning approaches used and the optimization space it considers, but also on the quality of training examples given to it. Good quality training examples should be both *representative* and *useful*, which means that they shall be sufficiently similar to the programs that the compiler may encounter in the future, and that they shall demonstrate significant performance improvement if proper transformations are applied. However, if the new programs are not similar enough to the training examples, the compiler might not be able to draw good enough heuristics from such so-called *outliers*. For instance, Fig.1 illustrates six programs (namely A, B, C, D, X and Y) in a feature space. Since A, B, C and D are close (and therefore similar) to each other, the compiler may directly apply to any one of them its optimization experience learned from the other three. However, its experience with these four programs might not be applicable or suitable for X and Y because they are considered outliers (since both X and Y are some distances away from the other four programs). In addition, if the training examples do not provide heuristics to what transformations to apply in order to achieve performance improvement, they would be of little use. In both cases, the effectiveness of learning-based optimization remains doubtful.

This suggests that training example selection plays a decisive part in the success of learning-based optimization. Most prior works this area assume that the compiler has been properly trained with good quality examples, which well suit the programs to be encountered. Few have properly addressed the issue of how these examples should be selected. Instead, they pick those examples manually, based on the developers' expertise.

3 A Training Example Selection Independent Approach

Learning based optimizing compiler should be *sustainable* if we aim to put it into practice. By *sustainable*, we mean that its effectiveness should be independent of the training examples provided, and evolve with more programs encountered. Therefore, a solution must be found for the above training example pitfall. To the best of our knowledge, this has not been properly addressed yet.

3.1 A Sustainable Learning-Based Compilation Paradigm

There are two approaches to tackle this training example pitfall. First, we could develop algorithms capable of automatically generating or selecting good-quality training examples for a learning compiler. Prior works[2,8] has provided advices on how to identify good transformation sequences from the given training examples. However, the fact that a compiler could encounter any arbitrary program suggests that the candidate program space is boundless. It is therefore hard to predict whether a training example is representative-enough unless we know in advance what programs will be encountered. Alternatively, we could develop techniques which enable the compiler to both deal with arbitrary programs given in arbitrary order and accumulate its experience from optimizing them in an iterative manner. When a program is encountered, the compiler first decides whether its prior optimization experience is applicable. If yes, then applies it directly, otherwise it turns to iterative exploration of the optimization space instead. Time and resource permitted, the search could last for as many rounds as necessary.

Although the latter approach might not promise instant success, we believe it provides more sustainable performance improvement over time. Furthermore, a compiler can therefore distinguish both less representative and less useful examples when more transformations are tested and more programs are encountered. These examples could either be transferred to a second-level knowledge-base or simply be abandoned when necessary, in order to keep the training set compact whilst maintaining a reasonable efficiency. This in return will benefit the learning approach. More importantly, such an evolutional approach will lead to a more sustainable compiler whose performance is independent of the training examples given.

We propose in this paper the idea of using reverse K-nearest neighbor (RKNN) algorithm to achieve a sustainable learning-based compiler, whose performance evolves with more programs encountered but is independent of the training examples provided. Given arbitrary programs, it can decide, as illustrated in Fig.2, whether it shall explore from scratch a predefined optimization space composed of various transformations, or directly apply its experience learned from other programs it has optimized before. This decision is based on its observation on both the new program and those it has optimized before, particularly the similarities in between. It is worth noting that the learning-based branch is not necessarily a one-off as expected, as some learning approaches might not be able to provide explicit optimizations instructions directly. Instead, the experience may come in the form of heuristics about some promising regions within the whole optimization space, as some prior related works[8] demonstrated.

This sustainable learning based compilation process (as illustrated in Fig.2) is just an extension to that presented in [6]. The newly added path (indicated by the grey boxes) indicates that, if the compiler has previously optimized programs sufficiently similar to the new one, it can directly apply its experience (as shown by the three steps followed), instead of heading for iterative search. This grey branch stands for the standard learning-based optimization process suggested by many prior works.

The only problem remained is to decide on whether this new program is sufficiently similar to programs encountered before. We use reverse K-nearest neighbors algorithm to solve this problem, as explained below.

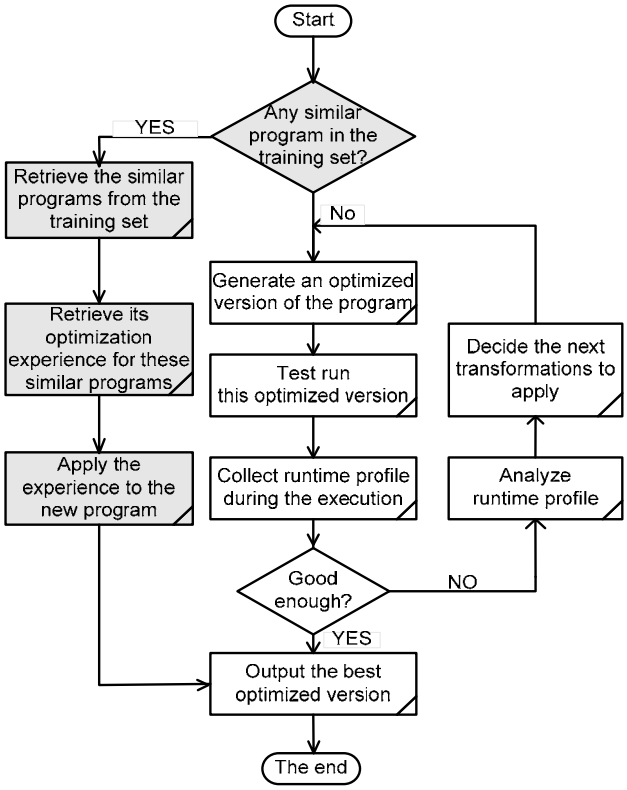

Fig.2. The sustainable learning-based optimization process proposed

3.2 Reverse K-Nearest Neighbor Algorithm

Reverse K-nearest neighbors (RKNN) algorithm[7] is considered as a complementary to the K-nearest neighbors (KNN) algorithm[14], which is a simple instance-based learning approach used for classifying object based on closest training examples in the feature space. It is worth noting that the nearest neighbor relation is not symmetric (i.e. the set of points closest to a query point is different from the set of points that have the query point as their nearest neighbors). RKNN aims to identify the influence of a query object on the whole data set by classifying it via a majority vote of its *K* nearest neighbors.

KNN considers all training examples as vectors in a multi-dimensional feature space, each with a class label. Euclidean distance is usually used to calculate the distance between vectors. Given a query q, RKNN retrieves all the points $p \in \mathrm{P}$ that have q as one of their k nearest neighbors. Specifically, $\mathrm{RKNN}(q) = \{p \in \mathrm{P} \mid \mathrm{dist}(p,q) \leqslant \mathrm{dist}(p,p_K)\}$, where p_K is the K-th farthest nearest neighbor of p. Intuitively, RKNN algorithm extends KNN by returning all objects that have q among their actual K-nearest neighbors. Given the query/object q and a database D of n objects, the set of K-nearest neighbors of q is the smallest set S that contains at least k objects from D such that the distance between q and any object o within S is less than that between q and

any object *o'* within *D-S*. The set of reverse *K*-nearest neighbors of *q* is therefore defined as the join set of all the objects which have *q* in their *K*-nearest neighbor sets. Various weights could be given to different dimensions in order to achieve better classification by the use of evolutionary algorithms or mutual information of the training data with the training classes.

Take the six programs illustrated in Fig.1 as an example. Both KNN and RKNN can correctly group A, B, C and D as near neighbors for each other. However, KNN still considers Y and D as X's nearest neighbors, because there is no other program closer to it. Even A, B and C will be included in X's *K*-nearest neighbors if *K* is greater than 2. This will result in the compiler applying to X improper/unsuitable optimization heuristics learned from them. However, with RKNN, the compiler can correctly identify X and Y as unfamiliar outliers and launch iterative search for them instead.

The naive RKNN algorithm is a computational intensive one, due to the cost of KNN algorithm, especially when the training set grows large. In addition, KNN tends to let the class with the most frequent examples dominate the prediction of a new object due to the majority voting mechanism it uses. Furthermore，the prediction accuracy could be degraded by noisy or irrelevant features.

3.3 RKNN in Learning-Based Optimization

Ideally, a sustainable learning-based compiler does not need an explicit training process before being put into practice. Instead, it will include in its training set all the programs it has encountered since its deployment. When a new program is encountered, the compiler shall decide at first whether it has sufficient prior experience to deal with it. If yes, it may directly apply its experience. Otherwise it shall turn to the iterative optimization instead. This decision is based on the set of programs it has encountered before, i.e. whether or not this new program is sufficiently similar to anyone within this set. In addition, its optimization experience of this new program shall also be added to this set. Therefore, with more and more programs encountered, the compiler accumulates its experience with an ever-growing training set.

We call a program an *outlier* if it is not sufficiently similar to anyone the compiler has encountered before. Prior works in many areas show that RKNN algorithm is effective in outlier detection. We therefore use it to detect whether a program encountered is an outlier or not, so that the compiler can take proper action accordingly. The algorithm is outlined in Fig.3.

We develop a data structure to support the above algorithm in order to ensure its efficiency when more programs are encountered. Since each feature vector *FV* in list *L* represents a program in the training set, each *FV* is attached with a list of its *K*-nearest neighbors. Each element of this attached KNN list is a <*ref*, *dist*> pair, where *ref* is a reference to a program in FV's k-nearest neighborhood, and *dist* the corresponding distance. This KNN list is kept sorted in either ascending or descending order of these *dist*s so that, if a new program *PV* is found closer to *FV* than anyone in this KNN list of *FV*, it can easily replace the far-most one in the list. For the new program *PV*, its KNN list is also constructed in the same manner when its distances with all the FVs are calculated during the process in Fig.3.

```
// given a list L of program feature vectors FVis, each
// stands for a program the compiler has encountered
// given a new program P
Capture the features of the new program P
Construct its feature vector PV
Set S to be empty  // let S be PV's RKNN set
for (each FVi in L) {
    calculate d, the distance between PV and FVi
    if  (d is shorter than that of any of FVi's KNN')
    {   // PV is considered among FVi's KNN
        Replace FVi's farthest neighbor with PV
        add FVi to S
}   }
Add PV to L
If  (S is empty) { Program P is considered an outlier }
```

Fig.3. Pseudo code for outlier detection via RKNN

Different features may have different ranges of raw values, therefore they shall first be unified before similarity calculation. A simple approach is adopted which uses the ratios instead of the raw values of all the features. Given two nodes A and B of feature vectors $<A_0, \ldots A_n>$ and $<B_0, \ldots B_n>$ respectively, their similarity is calculated as below,

$$\text{Similarity}(A, B) = \sum_{i=0}^{n} (w_i \times (\max(A_i, B_i) / \min(A_i, B_i)) \quad (1)$$

where w_i are weights given to different features according to their significance, and thresholds t_is are predefined for all the features we considered. For instance, given a threshold t=1.6, two loops of sizes 300 and 400 are considered similar because 400/300=1.33<1.6, whilst those of 400 and 200 are not as 400/200=2>1.6. More elaborate algorithms could be adopted for more accurate classification.

However, two loops of sizes 4000 and 3000 will still be considered similar although they are far apart in fact. A *valid distance* for each feature (for instance 100 for loop size) is introduced to tackle this problem. These valid distances define a *valid range* for each program/point P, where other points (Q_0, Q_1, …) within this range are considered candidates for neighborhood, and those beyond are not (even if P has no neighbor so far). It is worth noting that this valid distance is not considered as a predefined constant threshold. Instead, it is given an initial value when the compiler is first deployed. When more programs are encountered, the median of *P*'s valid neighbors (maximum *K*) can be obtained from the above data structure used. The *valid distance* will be therefore periodically updated to be the average of these medians (of all *P*s) so that it could be shortened with closer examples encountered. Intuitively, this means that, with more training examples, the experience obtained from each example will turn more specific for only certain types of similar programs.

Once the RKNN set of program P is decided, the compiler adopts a simple ballot mechanism to select transformations from its prior experience with programs in this set. The transformations voted by (i.e. appear in) the majority of programs in this RKNN set will then be applied to P in the order that wins the ballot. Further improvements of this mechanism will be made in our future work.

Table 1. Loop-level program features used

Loop Features
For loop is simple?
Loop nest depth
For loop has constant lower and upper bounds?
For loop has constant stride?
Size of the outmost loop
Number of array references in loop
Number of instructions in loop
Number of floating point variables in loop
Number of integer variables in loop
Loop contains an if-construct?
Loop iterator an array index?
All loop indices are constant?
Loop has branches?
Array accessed in linear manner?
Loop Strides on leading array dimensions only?

Table 2. Source-to-source transformations used

Loop Features
Loop unrolling (with factors 2, 3, 4)
Loop tiling (with sizes 3, 4, 5, 6)
Array padding (with size 2, 4)
Hoisting of loop invariants
Copy propagation
Common sub-expression elimination
Constant propagation
Dead code elimination

4 Preliminary Experiments and Results

In order to demonstrate the effectiveness of our proposed RKNN-based approach, we have carried out some preliminary experiments, whose details are specified below, before the results being presented and analyzed. For simplicity concern, we consider only 5 of each program's neighbors, i.e. K=5 in our experiments.

4.1 Program Features and Transformations

We consider fifteen loop-level features (as listed in Table.1) which we believe shall be sufficient to capture program characteristics. These features are chosen from [1] and listed in Table.1. They are considered equally important in our preliminary experiments. A source-level feature extractor is developed for our experiments.

Eight source-to-source transformations are considered in our preliminary experiments, as listed in Table.2. Considering the parameters for three of these transformations (loop unrolling, tiling and array padding), we have 14 different transformations in total.

It is worth noting that the RKNN approach we propose in this paper is independent of not only the program features but also the transformations considered here.

A random search algorithm was developed for the compiler to search within the resulting optimization space once it identifies a given program as an outlier. To generate a transformation sequence, this algorithm first randomly decides a sequence length (maximum 14), then creates the sequence by filling it with transformations randomly chosen from Table.2, The resulting sequence is then submitted to the compiler for evaluation. Considering the fact that many prior work in optimization space exploration[2][6] show that random search is capable of identifying good points quickly, we stop the search after 30 iterations. In addition, we also use this random search algorithm to determine the best performance improvement for each program in our experiments, in which case the search will last for 60 iterations.

In our experiments, we simplified the above ballot mechanism into an instance-based learning one. For each non-outlier *N*, its nearest neighbor *N'* is identified and the three transformation sequences which yield the top three speedups for *N'* are selected for *N*. The average of their performance achievements are considered as the final result.

4.2 Experimental Setup and Methodology

To keep our experiments at a reasonable and traceable scope, we chose benchmark programs from UTDSP[10] kernels (*fft*, *fir*, *iir*, *latnrm*, *mult* and *lmsfir*) and some Livermore kernels from BenchNT Classic Numeric[12] benchmark suites. We generated for each kernel multiple versions of various data sizes, as listed in Table.3.

Prior related works indicate that search-based algorithm can quickly identify points of significant speedups within a reasonable number of iterations, and not much further improvement can be achieved afterward. Therefore, we first applied the random search algorithm given in the previous subsection on each of these 43 selected kernels, in order to find out their best performance improvements. This search process is repeated for 60 iterations for each benchmark kernel *B* (as explained above), and the highest speedups achieved P_B is identified. This is repeated for five times for each kernel *B*, and considered the average of $P_{B,i}$ (i $\in$[1,5]) as the best performance achievable for *B*, B_{best}, as shown in the Highest Speedup columns in Table.3.

Next, we randomly generated 100 different submission sequences, each of which specifies a different order in which these 43 kernels are submitted for compilation. Next, each sequence *SubSeq* was tested for five times in order to minimize the noise impact. Similarly, we consider the final performance of each kernel program *B* in position *i* of *SubSeq* as the average of its performance over these five tests. This performance $P_{B,SubSeq,i}$ is compared against the B_{best} collected in the previous step, which indicates the effectiveness of the proposed approach.

Table 3. The 43 kernels and their highest speedup achieved after 60 iterations of random search (i.e. Best of Random Search as in Figure.4)

Kernel	Data Size	Highest Speedup	Kernel	Data Size	Highest Speedup
fft	_256	1.41	kernel7	_1000	1.05
	_1024	1.65		_2000	1.03
fir	_32_1	1.54		_3000	1.05
	_256_64	1.61	kernel8	_1000	1.23
iir	_1_1	1.21		_2000	1.21
	_4_64	1.19		_3000	1.27
latnrm	_8_1	1.34	kernel9	_1000	1.15
	_32_64	1.37		_2000	1.16
lmsfir	_8_1	1.39		_3000	1.21
	_32_64	1.44	kernel12	_1000	1.07
mult	_100	1.21		_2000	1.05
	_200	1.32		_3000	1.08
	_300	1.31	kernel13	_1000	1.04
	_400	1.45		_2000	1.06
	_500	1.56		_3000	1.09
	_1000	1.51	kernel22	_1000	1.14
kernel3	_1000	1.09		_2000	1.14
	_2000	1.12		_3000	1.16
	_3000	1.10	kernel24	_1000	1.21
kernel5	_1000	1.11		_2000	1.17
	_2000	1.14		_3000	1.20
	_3000	1.13			

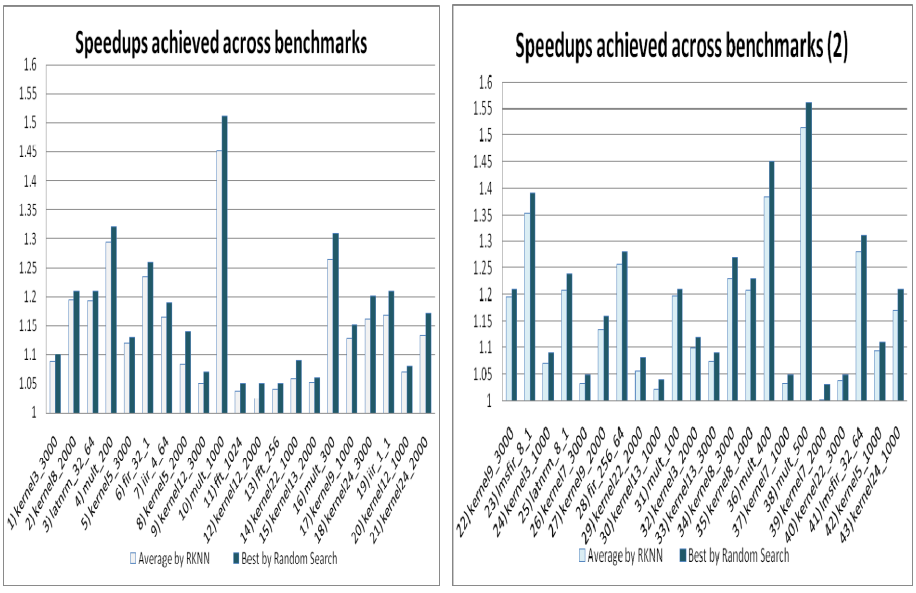

Fig.4. Experimental results for one randomly generated submission sequence. The *x*-axis lists the benchmark kernels and the *y*-axis is the speedups achieved (with baseline speedup=1).

The experiments were carried out on an AMD platform which has a 1GHz AMD Athlon(tm) 64 X2 Dual Core Processor 3600+ and 1G RAM, with gcc x86_64-linux-gnu 4.3 (with –O3 compile flag) running under Ubuntu Linux 4.3.2-1ubuntu12.

4.3 Results and Analysis

Some experimental results are presented in Fig.4, which shows the performance improvements found for each benchmark program in one randomly generated submission sequence *SubSeq*. These programs and their positions in the sequence are given in the x-axis, with the corresponding speedups achieved indicated in the y-axis. For instance, "*1) kernel3_3000*" stands for *kernel3_3000* at the 1st position of this *SubSeq*.

For the programs at the start of *SubSeq* (i.e. *kernel3_3000, kernel8_2000, latnrm_32_64, mult_200, kernel5_3000, fir_32_1* and *iir_4_64*), the compiler considered them outliers and applied the random search algorithm within the predefined transformation space and found most of the performance improvements within 30 attempts, 90% on average.

For program *kernel5_2000* at the 8th position of *SubSeq*, the compiler found it very similar to *kernel5_3000* encountered before (only differ in feature *Size of the outmost loop*). Therefore the compiler applied the best transformation learned from *kernel5_3000* and obtained average speedups of 1.10, 1.05, 1.09, 1.08, 1.10 in its five attempts. The average speedup achieved is 1.08, compared to that of 1.14 obtained via search. Similar results can be found for many of the following kernels in *SubSeq*, for instance *kernel12_3000* in the 9th position, *kernel9_3000* and *_2000* at the 22nd and 27th of *SubSeq*, etc. For *kernel12_3000* at the 9th position of *SubSeq*, the compiler judged that it is similar to kernel3_3000 at the 1st position, Therefore, it applied the experiences learned from the latter to *kernel12_3000*, and achieved speedups of 1.04, 1.05, 1.05, 1.06 and 1.05 respectively, i.e. an average speedup of 1.05 over five attempts, compared to the max speedup of 1.07 achieved via random search.

It is worth noting that in the 12th position of *SubSeq*, *kernel12_2000* was optimized by the compiler with the experience learned from *kernel12_3000* which, in turn, has been previously optimized with the experience learned from *kernel3_3000*, as discussed above. Such an indirect experience results in a relatively modest speedup achieved in 1.024 on average. In contrast, the random search algorithm achieved 1.05 on average after 60 attempts. However, in another submission sequence where *kernel12_3000* is positioned before *kernel12_2000*, the compiler used random search for the former and achieved a higher speedup of 1.06 (1.07 the highest). This direct experience was then applied to *kernel12_2000*, and the resulting speedup was 1.04. This suggests that direct experiences are more valuable than those learned indirectly from a less similar program. Therefore, it is suggested that the compiler shall pursue for direct, first-hand optimization experience when time and resources permitted.

Kernel7_3000 at the 26th position of *SubSeq* is very similar to many of the programs (for instance *kernel5_3000* and *kernel5_3000*) the compiler have encountered before. It was therefore optimized by experiences learned from the others.

Table.4. Performance of our proposed approach on the 15 benchmark programs across different data sizes and 100 submission sequences. The percentages given here are obtained by first calculating the ratio of the speedups achieved against the highest speedups for all different data sizes, before averaging them for each benchmark.

Kernel	Positions in the submission sequences			
	1-10	11-20	21-30	31-43
fft	89%	80%	82%	84%
fir	87%	82%	85%	86%
iir	90%	82%	84%	87%
latnrm	91%	84%	87%	90%
lmsfir	93%	86%	86%	86%
mult	94%	88%	90%	93%
kernel3	92%	85%	88%	91%
kernel5	92%	82%	84%	88%
kernel7	88%	74%	75%	80%
kernel8	90%	78%	84%	90%
kernel9	88%	80%	82%	85%
kernel12	79%	75%	80%	83%
kernel13	85%	70%	76%	85%
kernel22	89%	84%	85%	88%
kernel24	94%	84%	88%	92%
Average	*89%*	*81%*	*84%*	*87%*

However, due to the function call of exp in *kernel7*, these experiences achieved relatively modest speedups in 1.03 on average, i.e. 60% of the max speedup of 1.05.

Given this submission sequence *SubSeq* of these 43 benchmark kernels, we achieved 90% of the max speedups or higher for 12 of them (*kernel8_2000, latnrm_32_64, mult_200, kernel5_3000, fir_32_1, kernel_9_3000, lmsfir_8_1, fir_256_64, mult_100, kernel8_1000, mult_500* and *lmsfir_32_64*). For the majority of these 12 kernels, the performance improvements come from random search when the compiler considered them as outliers, particularly at the starting stage of the compiler, as discussed above. It achieved 80% of the max speedups or higher for 18 others (*kernel3_3000, iir_4_64, mult_1000, fft_256, kernel13_2000, mult_300, kernel9_1000, kernel24_3000, iir_1_1, kernel12_1000, latnrm_8_1, kernel9_2000, kernel3_2000, kernel13_3000, kernel8_3000, mult_400, kernel5_1000* and *kernel24_1000*). For the rest, we achieved 70% of the max speedups or higher for six (*kernel12_3000, fft_1024, fir_256_64, kernel_24_2000, kernel22_2000*, and *kernel22_3000*), and less than 70% for the remaining seven. On average, 80% of the highest performance improvement across all 43 benchmark kernels via this *SubSeq*. Similar results can be obtained via other submission sequences.

Table.4 summarizes the average performance of our proposed approach on the 15 benchmark programs over the 100 submission sequences we have tested in our experiments. For instance, *kernel12_1000, kernel12_2000* and *kernel12_3000* achieved only 79% of their respective highest speedups when they appeared between 1st and 10th positions within these 100 submission sequences. This is because only modest speedups of 1.07, 1.05 and 1.08 can be achieved for them within the given optimization space, and therefore the noise disturbance appears more significant than in other cases, which lead to the relatively modest efficiency.

It is worth noting that the majority of these 43 benchmark programs achieved on average 89% of their performance improvements when appeared at the 1st quarter (from 1st to 10th) of these 100 sequences randomly generated. This is mainly because the compiler had little experience when first launched. It considered these programs as outliers, and made a random search to optimize them. Therefore, most of the speedups achievable were obtained after 30 iterations. However, when these programs appeared in the 2nd quarter (11th to 20th) of the submission sequences, the compiler had already gained some experiences from optimizing other programs. However, such experiences were not sufficient to provide good enough hints for them (as detailed analysis of the raw results demonstrate), which results in only 81% of the max speedups on average.

With more programs optimized in the 1st and 2nd quarters of the submission sequences, the compiler became better equipped with more useful experiences learned from them, and therefore is capable of making better decisions for programs encountered in the following quarters. This results in higher performance achieved for these programs when they appeared in the 3rd and 4th quarters of these 100 sequences, as shown in Table.4. On average, we achieved 89%, 81%, 84% and 87% for these 43 benchmarks programs when they are positioned in different quarters of the 100 submission sequences. This demonstrates the effectiveness of the RKNN-based outlier detection approach and the corresponding optimization approach we proposed.

5 Related Work

Various approaches have been developed to explore an optimization spaces composed of various transformations. Early work by Kisuki et.al [6] uses random and grid-based search to explore a small optimization space composed of arbitrary combinations of parameterized loop transformations. Almagor et.al [2] randomly select transformations before applying them to the given programs in a global manner. Runtime feedbacks are then used to build a probabilistic model to guide transformation selection. No mechanism is provided either to avoid interferences between different transformations or to keep and accumulate optimization experience. Trianatafyllis et.al [17] use developers' experience to eliminate from an optimization space subspaces less likely to provide performance improvement. Kulkarni et.al [8] use genetic algorithm to search for more efficient transformation sequences based on specified fitness criteria. Franke et.al [5] develop two different probabilistic approaches, which compete against each other during the exploration of a space composed on 81 transformations, before the best sequence is identified at the end. This balances the tradeoff between search efficiency and coverage.

Most works on learning based compilation focus on how to accelerate the iterative optimization process, i.e. to reduce the number of "transform, compile and execute" iterations needed. Agakov et.al [1] uses predictive modeling to help a compiler to focus its optimization efforts on promising subspace of a given optimization space. Dubach et.al [4] predict the effect of different transformations on a given program via a performance model, which correlates code features and performance of a small number of its variants. Our proposed approach shares a common goal in elimination of explicit training, but differs in trying to decide whether prior experience can be applied to a

given new program, whilst their approach tries to build performance model for each given program.

Learning has also been successfully applied to parallelizing compilers. For instance, Long et.al [11] develops a cost-aware workload allocation mechanism for Java multi-threading. Wang et.al [18] proposes two predictors to decide the number of threads to use and the best scheduling policy for a given multi-core platform. Tournavitis et.al [16] uses machine learning techniques to make better mapping decision and provide more scope for adaptation to different target architectures. Chen et.al [3] proposed an adaptive OpenMP-based mechanism, which can generate a reasonable number of representative multi-threaded versions for a given loop on a given multi-core architecture. It then uses KNN algorithm to select at runtime a suitable version to execute based on runtime profile collected. A similar multi-versioning framework can also be found in [13].

Program/code features play a vital role in learning-based optimization. Leather et.al [9] present a mechanism to automatically identify features which most improve the quality of machine learning heuristics used in optimizing compilers.

6 Conclusion

We propose in this paper the idea of using reverse K-nearest neighbors approach to address the pitfalls of current learning-based optimization approaches, particularly its reliance on training examples. This results in a sustainable optimizing compiler which, when given arbitrary programs, can decide whether to explore from scratch a predefined optimization space, or to directly apply its experience learned before.

Tests of this approach with larger benchmark suites are currently underway. We also plan to introduce a larger optimization space with more transformations in order to see more significant achievement of our approach. Empirical Study of experimental results may help to evaluate the influence of the value of K on the “go search or apply experience” decision as well as its impact on final results. Further improvement can also be made in areas such as distance calculation and pruning outdated and less-useful experience, etc.

References

1. F.Agakov, E.Bonilla, J.Cavazos, et.al. Using machine learning to focus iterative optimization. Proc. of the 2006 International Symposium on Code Generation and Optimization (CGO’06), 2006
2. L.Almagor, K.Cooper, A.Grosul et.al. Finding effective compliation sequences. Proc. of ACM SIGPLAN 2003 Conference on Languages, Compilers and Tools for Embedded Systems. 2004.
3. X.Chen and S.Long. Multi-versioning for OpenMP parallelization via machine learning, The 2009 IEEE International Workshop on Multi‐Core Software Systems (in conjunction with The 15th International Conference on Parallel and Distributed Systems), 2009.

4. C.Dubach, J.Cavazos, B.Franke et.al. Fast compiler optimization evaluation using code feature based performance prediction. Proc. of the 4th International Conference on Computing Frontiers (CF'07), 2007.
5. B.Franke, M.O'Boyle, J.Thomson, et.al. Probablistic source-level optimization of embedded programs. Proc. of the ACM SIGPLAN 2005 Conference on Languages, Compilers and Tools for Embedded Systems, 2005.
6. T.Kisuki, P.Knijnenburg and M.O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. Proc. of the 2000 International Conference on Languages and Compilers for Parallel Computing. 2000.
7. F.Korn and S.Muthukrishnan, Influence sets based on reverse nearest neighbor queries, Proc. of the 2000 ACM SIGMOD international conference on Management of data, 2000.
8. P.Kulkarni., W.Zhao, H.Moon, et al. Finding Effective Optimization Phase Sequence[A]. Proc. of ACM SIGPLAN 2003 Conference on Languages, Compilers and Tools for Embedded Systems, US:2003.
9. H.Leather, E.Bonilla and M.O'Boyle. Automatic feature generation for machine learning based optimizing compilation. Proc. of the 2009 International Symposiom on Code Generation and Optimization (CGO2009), 2009.
10. C.Lee. UTDSP benchmark suite. *http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html*, 1998.
11. S.Long, G.Fursin, B.Franke. A cost-aware parallel working allocation approach based on machine learning techniques, Proc. of IFIP International Conference on Network and Parallel Computing, 2007.
12. R.Longbottom. BenchNT Classic Numeric benchmark suite. *http://www.roylongbottom.org.uk/*, 2010.
13. L.Luo, Y.Chen, C.Wu, S.Long, et.al. Finding representative sets of optimizations for adaptive multiversioning applications, Proc. of the 3rd Workshop on Statistical and Machine learning approaches to Architecture and compilaTion (SMART'09), 2009
14. T.Mitchell. Machine learning. US: McGraw-Hill Press, 1997.
15. J.Thomson, M.O'Boyle, G.Fursin et.al. Reducing Training Time and Calculating Confidence in a Machine Learning-based Compiler. Proc. of 22nd International Workshop on Languages and Compilers for Parallel Computers (LCPC'09), 2009.
16. G.Tournavitis, Z.Wang, B.Franke, et.al. Towards a holistic approach to auto-parallelization, integrating profile-driven prallelism detection and machine learning based mapping. Proc. of ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009), 2009.
17. S.Trianatafyllis, M.Vachharajani, N.Vachharajani, et.al. Compiler optimization space exploration. Proc. of the ACM/IEEE 2003 International Symposium on Code Generation and Optimization, 2003.
18. Z.Wang and M.O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. Proc. of 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009.