An Improved Machine Learning Approach for Selecting a Polyhedral Model Transformation

Ray Ruvinskiy and Peter van Beek

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract. Algorithms in fields like image manipulation, signal processing, and statistics frequently employ tight CPU-bound loops, whose performance is highly dependent on efficient utilization of the CPU and memory bus. The *polyhedral model* allows the automatic generation of loop nest transformations that are semantically equivalent to the original. The challenge, however, is to select the transformation that gives the highest performance on a given architecture. In this paper, we present an improved machine learning approach to select the best transformation. Our approach can be used as a stand-alone method that yields accuracy comparable to the best previous approach but offers a substantially faster selection process. As well, our approach can be combined with the best previous approach into a higher level selection process that is more accurate than either method alone. Compared to prior work, the key distinguishing characteristics to our approach are formulating the problem as a classification problem rather than a regression problem, using static structural features in addition to dynamic performance counter features, performing feature selection, and using ensemble methods to boost the performance of the classifier.

1 Introduction

Loops are a fundamental part of many scientific computing algorithms, with applications in image manipulation, sound and signal processing, and statistical simulations, among others. Loops in such algorithms are usually *tight*—they are computationally intensive and run for many iterations without blocking to perform high-latency operations such as disk or network I/O.

To achieve high performance, such loops must be structured so as to effectively utilize architectural features including instruction level parallelism, branch prediction, instruction and data prefetch capabilities, and multiple cores. Manually transforming the code to take advantage of architectural features results in code that is difficult to read and maintain. It also requires CPU architecture expertise that an application developer may not have. Previous work has focused on methods for automatically selecting transformations that preserve the semantics of the original code while optimizing the runtime of the loop nest on a given CPU architecture (see the description of related work in Section 3).

We focus on the polyhedral model, a mathematical framework wherein loop nests can be represented with polyhedra and loop transformations are expressed as algebraic operations on polyhedra [14,15]. This enables the generation of a search space of loop transformations that preserve the semantics of the original loop. Park et al. [13] use supervised machine learning to select a transformation from the overall transformation space, using as features hardware performance counter values obtained while profiling the candidate program. In particular, they use regression models to predict the speed-up of an arbitrary transformation from the search space over the original program. Park et al. propose a *five-shot approach*, where binaries generated by applying each of the best five transformations suggested by the model are run, and the transformation that has the fastest runtime is chosen. Their five-shot approach, however, can result in an excessively long selection phase.

We present an improved machine learning approach that achieves performance comparable to the best results reported by Park et al. without the need for a five-shot selection process—it is sufficient to use the single transformation selected by our classifier—and the selection process is substantially faster. As well, our approach can be combined with Park et al.'s five-shot approach into a higher level *six*-shot selection process that is more accurate than either method alone. The key contributing factors to the improvements offered by our method include: formulating the problem as a classification problem rather than a regression problem, using static structural features in addition to dynamic performance counter features, performing feature selection, and using ensemble methods to boost the performance of the classifier.

2 Background

In this section, we review the necessary background in the polyhedral model. For more background on this topic, see, for example, [15].

The polyhedral model is a mathematical framework used to represent loops as *polyhedra* and facilitate loop transformations [4,14,15]. Algebraically, a polyhedron encompasses a set of points in a $\mathbb{Z}^n$ vector space satisfying the inequalities,

$$\mathcal{D} = \{x \mid x \in \mathbb{Z}^n, Ax + a \geq 0\},$$

where A is a matrix while x and a are column vectors. In the polyhedral model, A is a matrix of constants, x is a vector of iteration variables, a is a vector of constants, 0 is the zero vector, and the inequality operator ($\geq$) compares the vectors element-wise.

To be mapped to the polyhedral model, a block of statements must be restricted such that it is only enclosed in *if* statements and *for* loops. Pointer arithmetic is disallowed (but array accesses are allowed). Function calls must be inlined and loop bounds are restricted to affine functions of loop iterators and global variables. While these limitations appear onerous at first glance, such computational kernels play a large role in scientific and signal processing [2,6].

Loop bounds determine the *iteration domains* of the statements in consideration. An iteration domain represents the values taken on by the loop iterators for all iteration instances. The following example shows a simple nested loop and its associated iteration domain.

Example 1. Given the source code,

```
int A[n][m];
int B[m][n];
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    if (i < j)
      A[i][j] = n*m*B[j][i];
```

the iteration domain $\mathcal{D}_S$ of statement S in the polyhedral model becomes,

$$\mathcal{D}_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 1 \\ n \\ 1 \\ m \\ 0 \end{pmatrix} > 0 \right\}.$$

The first two inequalities reflect the bounds imposed by the outer loop; the next two inequalities are the bounds imposed by the inner loop; and the last inequality reflects the logic of the conditional.

In the polyhedral model, a *timestamp* is associated with each statement instance. The relative order of the timestamps represents dependencies between the execution order of the instances. If two instances share a timestamp, no ordering between them need be enforced and they can be executed in parallel.

A multi-dimensional affine *schedule* specifies the execution order of each statement instance that respects the dependency information [14,15]. There are often many possible schedules, and each schedule, or polyhedral transformation, can be viewed as a program transformation. It is possible to algorithmically generate the set of schedules, where each schedule preserves the semantics of the original loop nest. Some schedules can be described as a sequence of well-known loop optimizations including: tiling [18], fusion [9], unrolling, prevectorization [10], and parallelization. However, there are also valid schedules that are not representable as a sequence of known loop optimizations.

Example 2. Consider the following semantically equivalent program transformation of the source code shown in Example 1,

```
for (c0 = 0; c0 <= n/32; c0++)
  for (c1 = 0; c1 <= m/32; c1++)
    for (i = 32*c0; i < min(n,32*(c0+1)); i++)
      for (j = 32*c1; j < min(m,32*(c1+1)); j++)
        if (i < j)
          A[i][j] = n*m*B[j][i];
```

The original source code shown in Example 1 could have poor cache behavior for larger values of n, possibly incurring a cache miss on every access of an element of matrix B. The transformed source code is an example of tiling. While more complicated, the transformed code can be algorithmically generated and could have significantly better cache performance.

Two schedules, or polyhedral transformations, can differ dramatically in performance on an architecture. However, selecting the appropriate polyhedral transformation is difficult even for machine architecture experts as the individual components can interact with each other and cannot be chosen independently.

3 Related Work

There are two main approaches to automatically selecting a high-performance polyhedral transformation: hand-crafted and machine learning approaches.

For hand-crafted approaches, Lim and Lam [11] propose an algorithm to select a transformation that maximizes parallelism and minimizes synchronization. However, their algorithm has super-exponential complexity in the worst case. Pouchet et al. [14,15] present search-based approaches to selecting a transformation: an exhaustive search and a genetic algorithm. Pouchet et al. [16] subsequently improve this approach to use a combination of analytic models, where they are available and effective, and empirical search for cases where analytic models do not account for important properties of the hardware that have a significant impact on performance. These search-based approaches all suffer from scalability concerns, as the size of the search space is large enough to make it impractical to explore the space iteratively and evaluate different transformations by compiling and running them.

For machine learning approaches, Park et al. [13] use regression models to predict the runtime of a polyhedral transformation of a source program and so select a transformation. The search space is limited to the transformations supported out-of-the-box by the Polyhedral Compiler Collection version 1.1 (PoCC) software: loop tiling, loop fusion, loop unrolling, loop prevectorization, and loop parallelization. Considering every combination of the transformations results in a search space of several hundred transformations. As in Cavazos et al. [3], Park et al. use hardware performance counters as features. Park et al. propose a technique called *five-shot* which requires running the program being optimized five times. For some programs their approach can be excessively time consuming, which may serve as a barrier to the uptake of the technique. Our approach builds on but improves Park et al. to either remove this deficiency or to improve the accuracy. In Section 5, we perform an extensive empirical comparison.

More recently, Park et al. [12] extend this work to include two additional loop optimizations—wavefront and SIMD vectorization—using a newer version of PoCC (version 1.2) and to propose a six-shot approach that consists of running the program six times, each time for a different regression classifier. Unfortunately, we were not successful at integrating version 1.2 into our experimental

setup and no experimental results comparing the six-shot to the original five-shot proposal have been reported. Our experimental results reproduce and compare against the original five-shot approach using version 1.1 of PoCC. However, as there is evidence that the six-shot approach only gives incrementally better performance over the five-shot approach, and the six-shot approach retains the same inherent deficiency, we believe our comparison and conclusions still hold.

Additionally, there is a considerable body of work on using machine learning more generally in optimization in compilers including selecting a loop unroll factor [17], selecting compiler optimization settings [3], and selecting a loop tiling size [20]. In our approach we adapt some of the features previously proposed for selecting loop tile size [20].

4 An Improved Machine Learning Approach

Our proposal is an extension of the work by Park et al. [13]. The key distinguishing characteristics to our approach compared to Park et al.'s work are using static structural features in addition to dynamic performance counter features (Section 4.1), formulating the problem as a classification problem rather than a regression problem (Section 4.2), performing feature selection to exclude features that hold little or no predictive value for the class (Section 4.4), and using ensemble methods to boost the performance of the classifier (Section 4.5).

4.1 Initial Feature Set

As in previous work, we use hardware performance counters as features [3,13]. Over 90 program counters are available on the architecture used in our experimental setup. We selected a subset of those we deemed to be promising (see Table 1). Hardware performance counter values are collected by running a binary compiled from the unmodified program source on representative input. These values are then normalized by taking ratios and manually discretized. We also added static memory access features that are extracted from the program source structure and reflect the program's memory access patterns, as proposed by Yuki et al. [20].

4.2 Class Value

Rather than using regression models to predict speed-up, we express the problem as a binary classification problem by considering which of two transformations results in faster runtime for a given source program. More formally, given a source program P and two transformations, T_A and T_B, the transformation pair (T_A, T_B) is labeled 1 if P has a faster runtime having had T_A applied rather than having had T_B applied; otherwise, the pair is labeled 0. A complete training example then consists of the feature values for the benchmark, the transformation pair, and the class of the transformation pair. A decision tree classifier is learned from the data.

Table 1: Performance counter features c_i and memory access features m_i.

feature	description
c_0	ratio of L1 cache misses to L1 cache accesses
c_1	ratio of L2 cache misses to L2 cache accesses
c_2	ratio of L3 cache misses to L3 cache accesses
c_3	ratio of total CPU cycles to retired instructions
c_4	ratio of cycles when no instructions were retired to total CPU cycles
c_5	ratio of L3 cache accesses to total CPU cycles
c_6	ratio of L3 cache misses to total CPU cycles
c_7	ratio of all resource-related stall cycles to total CPU cycles
c_8	ratio of load buffer stall cycles to total CPU cycles
c_9	ratio of stall cycles due to reservation station being full to total CPU cycles
c_{10}	ratio of stall cycles to reorder buffer being full
c_{11}	ratio of conditional branch instructions executed to retired instructions
c_{12}	ratio of mispredicted branches executed to retired instructions
m_0	number of prefetched memory reads
m_1	number of non-prefetched memory reads
m_2	number of loop-invariant memory reads
m_3	number of prefetched memory writes
m_4	number of non-prefetched memory writes
m_5	number of loop-invariant memory writes

Once the classifier is constructed, it is used to choose a polyhedral transformation as follows. For a previously unseen program P, the classifier labels all transformation pairs (T_A, T_B), for all transformations T_A and T_B in the transformation search space. Subsequently, for every transformation T_A, the number of times a tuple (T_A, T_B), where T_B is any transformation other than T_A, is predicted to have the label 1 is recorded as $score_{T_A}$. The transformation with the highest score is selected as the best transformation suggested by the classifier. In effect, a voting algorithm is used, with transformations being voted on within the context of every transformation pair combination. The transformation with the most votes is considered the winner. This is referred to as *pairwise preference ranking* or *round robin ranking* [5]. Other approaches to combining pairwise preferences exist (e.g., algorithms to calculate class probabilities [19]) but are not explored in the context of this paper.

4.3 Data Collection

Benchmarks from Polybench/C [1], a suite of computational kernels with loop nests, are used to train and test the classifier. The PAPI [2] library is used to collect performance counter values for each benchmark. Each benchmark is compiled using the `gcc` compiler version 4.7.2 at the highest standard optimization

[1] `http://www.cs.ucla.edu/~pouchet/software/polybench`
[2] `http://icl.cs.utk.edu/papi/index.html`

level, −O3. No other `gcc` flags are used. The benchmarks are then executed. The wall time runtime and selected hardware performance counter values are recorded. Each benchmark is run enough times for the sum of the runtimes of the executions to reach at least 10 seconds.

Subsequently, the Polyhedral Compiler Collection version 1.1 (PoCC) [3] is used to generate the transformation search space. PoCC has numerous options for various optimizations. We considered different values for the fusion, OpenMP, tiling, vectorization, and loop unrolling options. In addition, we included the identity transformation, where the original source code is not altered.

PoCC generates one source file per transformation per benchmark. The source files are compiled, generating a binary. The binary is executed and the wall time runtime is recorded. A training example is then generated for every pair of transformations in each benchmark. The format of a training example is c_0, ..., c_{12}, m_0, ..., m_5, $identity_A$, $tiling_A$, $openmp_A$, $vectorization_A$, $unroll_A$, $identity_B$, $tiling_B$, $openmp_B$, $vectorization_B$, $unroll_B$, *class*. The features c_0, ..., c_{12} are the discretized performance counter values for the benchmark, and m_0, ..., m_5 are the memory feature values for the benchmark. The features $identity_A$, ..., $unroll_A$, are the feature values for the first transformation in the pair, and the features $identity_B$, ..., $unroll_B$, are the feature values for the second transformation in the pair. *Identity* is a binary feature, and its value is 1 if the transformation is an identity transformation and 0 otherwise. *Tiling* is an enumerated feature corresponding to the tiling setting used in the transformation. The tiling setting consists of the tiling factors for the top three nested loops. Possible tiling factors are 1 (no tiling) or 32. This gives 8 values for the tiling setting in total. *Openmp* and *vectorization* are binary features reflecting their presence or absence in the transformation. The *unroll* value of unroll is the loop unrolling factor used (8, 4, or 0 for no unrolling). The feature *class* is the binary classification. If the runtime of the first transformation in the pair is less than the runtime of the second transformation, *class* is 1; otherwise, it is 0. The transformations T_A, T_B and T_B, T_A are treated as distinct pairs, and a separate training example is generated for each. The two pairs will belong to opposite classes.

4.4 Feature Selection

Feature selection (see, e.g., [7] and references therein) is used to select features that have predictive value. A classifier is generated using every combination of two features and evaluated for every benchmark. The classifiers are then sorted in order of the number of benchmarks where the classifier suggested a transformation with a better runtime than the expected runtime of a randomly selected transformation for the benchmark (see lines 2–8 of Algorithm 1). The random selection process is repeated 100 times. In contrast to Park et al. [12], who found that on average feature selection only degraded performance, feature selection is an integral component of our overall approach.

[3] `http://www.cs.ucla.edu/~pouchet/software/pocc`

Algorithm 1: Feature selection and classifier evaluation.

```
1  foreach b ∈ Benchmarks do
2      foreach f1, f2 ∈ Features do
3          foreach b′ ∈ (Benchmarks − {b}) do
4              Learn a classifier using f1, f2 and data from Benchmarks − {b, b′};
5              Test the classifier using data from b′;
6          end foreach
7          Tally number of benchmarks for which the classifier using the feature
           combination f1, f2 was judged effective;
8      end foreach
9      Sort feature combinations in descending order by number of benchmarks for
       which they were judged effective;
10     Take top 21 feature combinations and construct a decision tree from each
       combination using data from Benchmarks − {b};
11     Construct an ensemble classifier that classifies the data using the 21 trees
       and predicts the class by a majority vote of all 21 trees;
12     Use the ensemble classifier to predict best transformation for b;
13     Report the accuracy of the predicted transformation for b;
14 end foreach
```

The selected features and the number of times a feature occurred in a feature pair that was selected for the classifier ensemble are the following.

feature	c_0	c_1	c_3	c_4	c_5	c_6	c_7	c_{12}	m_1	m_3	m_4	m_5
number	8	4	3	1	6	2	1	3	4	3	2	1

The two features that appear the most often are the number of L1 cache misses normalized with respect to the number of L1 cache accesses and the number of L3 cache accesses normalized with respect to total CPU cycles. On the other hand, the number of L3 cache accesses normalized with respect to L3 cache misses does not appear at all. It is not immediately obvious why L1 cache misses are significant while L3 cache misses are not and why overall L3 cache accesses are significant while overall L1 cache accesses are not. This may be related to architectural peculiarities. It is also interesting to note that prefetched memory writes are significant, while prefetched memory reads are not. Again, this may be related to the architecture on which the experiments were run. Such subtleties regarding feature significance are not intuitive, and they become apparent only as a result of the feature selection process.

4.5 Classifier Ensembles

The performance of a classifier can often be boosted by generating multiple classifiers (or *base-learners* from smaller feature sets) and combining their results [1, pp. 419-421]. Such combinations of classifiers are known as *ensembles*. The simplest way to combine the outputs of an ensemble of classifiers is *voting*, and

the simplest and most widely-used voting technique is *simple voting*, where all classifiers are equally weighted [1, pp. 424-425].

A simple voting approach is used to construct a classifier ensemble for predicting the label of a transformation pair (T_A, T_B). The top n classifiers as determined in the feature selection stage vote to predict the class. (In addition to ensembles of decision trees containing pairs of features, we also considered ensembles of decision stumps but these did not lead to improved performance.) Each classifier's vote is weighted equally. A value of 21 is used for n. The value is determined empirically, and it is found that a value of 21 performs better than a lower value, while values between 21 and 40 all perform relatively equally well (see lines 9–11 of Algorithm 1).

5 Experimental Evaluation

We present an empirical evaluation of our proposal against a baseline and the current state-of-the-art approach. Our experimental results were obtained on an Intel Core i7: Nehalem microarchitecture, Lynnfield Performance Desktop, model 870, 4 cores, and clock rate of 2.93 GHz. We show that on a benchmark suite of computational kernels with loop nests, our method is competitive for accuracy (Section 5.1), offers a substantially faster selection process (Section 5.2), and can be combined with the best previous approach into a higher level selection process that is more accurate than either method alone (Section 5.3).

The results presented for linear regression (LR) and support vector machine regression (SVM), for both the one-shot and five-shot approaches, are our best efforts to reproduce the results reported in Park et al. [13]. One-shot results evaluate the transformation predicted to be the best by the given method. Five-shot results take the best runtime of the top five results as predicted by the given method; i.e., the binary corresponding to each of the top five results is run and the binary that has the best runtime is used.

5.1 Evaluation of Selection Accuracy

A nested leave-one-benchmark-out approach [8, pp. 245-247] was used to evaluate our overall approach (see Algorithm 1). For each benchmark, the data sets of all other benchmarks are used as the training data, while the data set of the benchmark in question is used as the test data. C4.5 [4] is used to generate decision trees from the training data.

Following Park et al., as a measure of accuracy we use percentage of optimal: the ratio of the runtime of the benchmark binary obtained by applying the optimal transformation in the search space to the runtime of a benchmark binary obtained by applying a transformation selected by the approach being evaluated. The ratio is expressed as a percentage, with a value of 100% indicating that the transformation selected is the optimal transformation in the search space. The

[4] `http://www.rulequest.com/Personal`

Table 2: Percentage of optimal on benchmark kernels for the identity transformation; and the transformations chosen by the linear regression (LR), support vector machine (SVM), and decision tree vote (DTV) methods.

benchmark	identity	LR		SVM		DTV
		1-shot	5-shot	1-shot	5-shot	
2mm	15.9	76.9	100.0	96.7	100.0	77.8
3mm	19.0	99.4	99.4	75.1	89.0	100.0
adi	73.3	46.5	54.1	40.8	48.8	100.0
bicg	73.6	68.1	68.1	33.7	100.0	100.0
cholesky	68.3	99.6	99.6	95.6	99.2	95.6
correlation	5.1	54.8	60.4	60.4	99.4	98.8
covariance	4.8	58.9	73.0	38.1	57.5	100.0
doitgen	46.6	32.0	67.7	66.6	100.0	31.7
durbin	70.5	98.7	99.4	98.6	98.7	97.9
dynprog	59.2	83.8	84.5	77.6	81.3	77.6
fdtd-2d	50.6	98.7	98.7	17.5	62.8	55.0
fdtd-apml	70.0	97.9	97.9	100.0	100.0	99.3
floyd-warshall	71.3	62.9	74.9	100.0	100.0	100.0
gemm	11.4	98.2	100.0	69.1	83.5	96.9
gemver	28.8	65.7	100.0	96.7	98.6	65.7
gesummv	73.7	66.8	67.0	100.0	100.0	83.7
gramschmidt	9.3	44.3	44.3	13.3	43.7	43.5
jacobi-1d-imper	77.2	93.8	94.4	94.7	99.1	99.1
jacobi-2d-imper	49.2	80.3	80.6	98.8	98.9	68.2
lu	65.7	78.5	79.0	41.5	41.5	100.0
ludcmp	69.4	83.6	99.6	95.9	99.6	86.0
mvt	31.9	98.0	98.0	90.8	99.6	53.3
reg_detect	70.4	86.7	99.7	98.2	98.2	98.2
seidel-2d	42.4	49.6	50.2	61.4	61.4	100.0
symm	70.0	98.4	98.5	97.3	98.3	98.5
syr2k	35.4	51.3	52.1	51.5	84.6	90.0
syrk	19.2	41.4	44.8	27.8	90.5	89.5
trisolv	80.8	57.3	57.7	62.6	100.0	100.0
trmm	25.2	57.4	58.5	35.9	100.0	90.6
average	47.9	73.4	79.4	70.2	87.4	86.1

optimal transformation is found by applying every transformation in the search space to the benchmark source code, running the resulting binaries, recording the runtimes, and selecting the transformation corresponding to the binary with the lowest runtime. As a baseline, we use the runtime of the stock benchmark with no modifications to the source code, referred to as `identity`.

Table 2 contrasts the accuracy, expressed as percentage-of-optimal, of the identity transformation, the one-shot and five-shot approaches for linear and SVM regression, and our decision tree voting approach. Table 3 shows the speed-up over the identity transformation for every benchmark for the five-shot SVM

Table 3: Speed-up on benchmark kernels over the identity transformation using the SVM five-shot (SVM) and decision tree vote (DTV) methods.

benchmark	SVM	DTV	benchmark	SVM	DTV
2mm	6.3	4.9	gesummv	1.4	1.4
3mm	4.7	5.3	gramschmidt	4.7	4.7
adi	0.7	1.4	jacobi-1d-imper	1.3	1.3
bicg	1.4	1.4	jacobi-2d-imper	2.0	1.4
cholesky	1.5	1.4	lu	0.6	1.5
correlation	19.4	19.3	ludcmp	1.4	1.2
covariance	11.9	20.7	mvt	3.1	1.7
doitgen	2.2	0.7	reg_detect	1.4	1.4
durbin	1.4	1.4	seidel-2d	1.5	2.4
dynprog	1.4	1.3	symm	1.4	1.4
fdtd-2d	1.2	1.1	syr2k	2.4	2.2
fdtd-apml	1.4	1.4	syrk	4.7	5.2
floyd-warshall	1.4	1.4	trisolv	1.2	1.2
gemm	7.3	8.5	trmm	4.0	3.6
gemver	3.4	2.3	median	1.5	1.4
			average	3.3	3.5

regression and the one-shot decision tree voting. The results of our reproduction of Park et al.'s work are similar to the results reported in their paper; thus, we have some confidence in their correctness. Our one-shot decision tree vote and the five-shot SVM approach perform about equally well.

5.2 Evaluation of Selection Speed

Table 4 contrasts the runtime (seconds) needed to select a transformation using the support vector machine and decision tree vote methods. For both methods, the untransformed code is first run to gather the feature values. These feature values are then fed into a classifier to predict the best transformations. Gathering the feature values is the most time consuming phase as enabling the program counter features slows the execution, in proportion to the number of program counters enabled. Park et al. [12] use 56 program counter features [5]. We effectively used feature selection to narrow this down to 8 program counters and 4 structural features. The speedups range from 3.7 to 15.8, with an average of 7.2. The maximum wall clock speed up was for the 3mm benchmark. Park et al. took 1154.3 seconds (approximately 18 minutes) whereas ours took 182.8 seconds (approximately 3 minutes), for a speedup ratio of six times faster.

[5] Park et al. [13] do not provide details on which program counters were used in their experiments, so we are relying instead on the expanded version of their paper [12] where they use 56 program counters, 47 of which are available on our architecture.

Table 4: Runtime (seconds) on benchmark kernels needed to select a transformation using the SVM five-shot (SVM) and decision tree vote (DTV) methods. Also shown is the ratio of the two runtimes. The average ratio is 7.2.

benchmark	SVM	DTV	ratio	benchmark	SVM	DTV	ratio
2mm	857.8	131.6	6.5	gesummv	20.4	3.3	6.2
3mm	1154.3	182.8	6.3	gramschmidt	487.1	75.9	6.4
adi	79.0	9.8	8.1	jacobi-1d-imper	20.7	4.0	5.2
bicg	17.6	2.5	7.0	jacobi-2d-imper	15.2	1.8	8.4
cholesky	15.4	1.9	9.1	lu	96.5	6.1	15.8
correlation	264.2	44.4	6.0	ludcmp	51.3	13.7	3.7
covariance	310.3	46.1	6.7	mvt	18.4	2.1	8.8
doitgen	68.4	10.3	6.6	reg_detect	17.8	1.8	9.9
durbin	50.3	8.2	6.1	seidel-2d	30.8	4.1	7.5
dynprog	72.8	10.6	6.9	symm	686.4	91.5	7.5
fdtd-2d	90.8	13.0	7.0	syr2k	222.3	33.4	6.7
fdtd-apml	74.7	9.1	8.2	syrk	163.1	25.0	6.5
floyd-warshall	81.3	9.8	8.3	trisolv	31.9	7.5	4.3
gemm	361.7	69.8	5.2	trmm	164.0	27.6	5.9
gemver	19.8	2.3	8.6				

5.3 A Six-Shot Selection Process

The per-benchmark breakdown in Table 2 reveals that while, on average, our one-shot DTV method is competitive with Park et al.'s five-shot SVM method, the different approaches do better on different benchmarks. For examples, DTV obtains 100% on `lu` when SVM obtains only 41.5%, and SVM obtains 100% on `doitgen` when DTV obtains only 31.7%.

That the two methods have different strengths suggests that combining them would be advantageous. We define a *six*-shot selection process which consists of: (i) run the five best transformations predicted by Park et al.'s SVM approach and record the best transformation, the transformed binary that has the fastest runtime, (ii) run the transformed binary predicted by our one-shot DTV approach, and (iii) report the fastest of the two transformations from steps (i) and (ii). The result is a six-shot selection process that is more accurate than either method alone: The accuracy is boosted from 87.4% and 86.1% for the SVM and DTV methods, respectively, to within 95.0% of optimal, on average. Note that adding step (ii) has a proportionally negligible effect on the overall runtime.

6 Conclusion

We presented an improved machine learning approach within the polyhedral framework to select the transformation of a loop nest that gives the highest performance on a given architecture. On a benchmark suite of computational kernels our DTV method achieves accuracy results competitive with Park et

al.'s [13] five-shot SVM regression approach, the best previous approach, while speeding up the selection process by a factor of 7.2 on average. As well, when the DTV approach is combined with Park et al.'s five-shot approach into a higher level six-shot selection process the new six-shot selection process is more accurate than either method alone. On the benchmark suite, the accuracy of the combined six-shot process, as measured by percentage from optimal, was boosted to 95.0% as compared to 86.1% and 87.4% when our method and Park et al.'s method, respectively, are used as stand-alone methods.

References

1. Alpaydin, E.: Introduction to Machine Learning. The MIT Press, 2nd ed. (2010)
2. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Proceedings of ETAPS'10/CC'10 (2010)
3. Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M.F.P., Temam, O.: Rapidly selecting good compiler optimizations using performance counters. In: Proceedings of CGO'07. pp. 185–197 (2007)
4. Feautrier, P.: Automatic parallelization in the polytope model. The Data-Parallel Programming Model, LNCS 1132, Springer. pp. 79-103 (1996)
5. Fürnkranz, J., Hüllermeier, E.: Pairwise preference learning and ranking. In: Proceedings of ECML-03. pp. 145–156 (2003)
6. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Intl J. of Parallel Programming 34, 2006 (2006)
7. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. J. Mach. Learn. Res. 3, 1157–1182 (2003)
8. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data mining, Inference and Prediction. Springer, 2nd ed. (2009)
9. Kennedy, K., McKinley, K.S.: Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Proc. of LCPC'94. pp. 301–320 (1994)
10. Larsen, S., Amarasinghe, S.: Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of PLDI'00. pp. 145–156 (2000)
11. Lim, A.W., Lam, M.S.: Maximizing parallelism and minimizing synchronization with affine transforms. In: Proceedings of POPL'97. pp. 201–214 (1997)
12. Park, E., Cavazos, J., Pouchet, L.N., Bastoul, C., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. Intl J. of Parallel Programming 41, 704–750 (2013)
13. Park, E., Pouche, L.N., Cavazos, J., Cohen, A., Sadayappan, P.: Predictive modeling in a polyhedral optimization space. In: Proc. of CGO'11. pp. 119–129 (2011)
14. Pouchet, L.N., Bastoul, C., Cohen, A., Cavazos, J.: Iterative optimization in the polyhedral model: Part II, multi-dimensional time. In: Proceedings of PLDI'08. pp. 90–100 (2008)
15. Pouchet, L.N., Bastoul, C., Cohen, A., Vasilache, N.: Iterative optimization in the polyhedral model: Part I, one-dimensional time. In: Proceedings of CGO'07. pp. 144–156 (2007)
16. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., Vasilache, N.: Loop transformations: convexity, pruning and optimization. SIGPLAN Not. 46, 549–562 (2011)

17. Stephenson, M., Amarasinghe, S.: Predicting unroll factors using supervised classification. In: Proceedings of CGO'05. pp. 123–134 (2005)
18. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: Proceedings of PLDI'91. pp. 30–44 (1991)
19. Wu, T.F., Lin, C.J., Weng, R.C.: Probability estimates for multi-class classification by pairwise coupling. J. Mach. Learn. Res. 5, 975–1005 (2004)
20. Yuki, T., Renganarayanan, L., Rajopadhye, S., Anderson, C., Eichenberger, A.E., O'Brien, K.: Automatic creation of tile size selection models. In: Proceedings of CGO'10. pp. 190–199 (2010)