Tile Size and Loop Order Selection using Machine Learning for Multi-/Many-Core Architectures

Shilpa Babalad
shilpab@iisc.ac.in
Indian Institute of Science
Bengaluru, Karnataka, India

Shirish K Shevade
shirish@iisc.ac.in
Indian Institute of Science
Bengaluru, Karnataka, India

Matthew Jacob Thazhuthaveetil
mjt@iisc.ac.in
Indian Institute of Science
Bengaluru, Karnataka, India

R Govindarajan
govind@iisc.ac.in
Indian Institute of Science
Bengaluru, Karnataka, India

ABSTRACT

Loop tiling and loop interchange (or permutation) are techniques that can expose task and data-level parallelisms and can exploit data locality available in multi-dimensional loop nests. Choosing the appropriate tile size and loop order is important to achieve significant performance improvement. However, the effect of these transformations on the performance of the loop nest is not straightforward due to the complex interplay of several architectural features in multi-/many-core architectures. In this work, we propose using a supervised learning technique and develop a Support Vector Machine (SVM) based hierarchical classifier to identify the best-performing tile size and loop order for a given loop nest. Our approach results in identifying tile sizes and loop orders whose performance, on average, is within 18% and 9% of the *optimal performance* for two sets of loop nests on Intel Xeon Cascadelake architecture. Further, our method outperforms state-of-the-art techniques, Pluto and Polly, with a geometric mean speedup of 1.35x to 1.58x.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; • **Computing methodologies → Classification and regression trees**; • **Software and its engineering → Compilers**.

KEYWORDS

Loop transformations, Vectorization and Parallelization, Supervised learning, Support Vector Machine, Hierarchical Classifier

ACM Reference Format:
Shilpa Babalad, Shirish K Shevade, Matthew Jacob Thazhuthaveetil, and R Govindarajan. 2024. Tile Size and Loop Order Selection using Machine Learning for Multi-/Many-Core Architectures. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656630

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656630

1 INTRODUCTION

Loops form the major compute-intensive and data-intensive part of many real-world applications. As per the 90-10 rule, these loops form only 10% of the application code but account for 90% of the application execution time. Hence, they require elaborate and extensive optimizations/transformations that, in turn, help in reducing the overall execution time of the application. The transformations are done either automatically by a compiler or manually by the programmers, help in exploiting the architectural features like memory hierarchy, multiple cores, vector processing units, hardware prefetchers present in modern multi-/many-core systems.

Loop tiling and loop interchange are important transformations to realize the high performance of an application consisting of multi-dimensional loops [6]. Tiling reorganizes the traversal of the iteration space of a loop to exploit parallelism and/or temporal and spatial locality. Tiling transformation involves techniques for identifying the size or shape of the tile and tiled code generation. Identifying the tile size that results in the lowest execution time is complex and involves developing either a heuristic approach or an autotuner to select the tile size, making use of a cost model. Constructing a good cost model that can represent the architecture under consideration is hard due to the interplay among multi-level cache hierarchies, hardware prefetch configurations, and optimization phases of a compiler.

Application of loop permutation/interchange transformation on a tiled code can result in multiple valid transformed loop orders [6]. The performance of a transformed loop depends on its ability to exploit multiple cores, vector processing units, multi-level cache hierarchies, and hardware prefetchers present on the multi-/many-core architectures. Existing polyhedral loop transformation techniques identify and expose different types of parallelism and data locality. They transform a given loop into a tiled loop with a legal loop order that satisfies all the data dependencies in the original loop. They often fail to pick the best-performing loop order.

We refer to the tile size and the loop order that results in the lowest execution time of a loop nest as the *best-performing* (tile size, loop order) combination. Informally, we also refer to these as best-performing tile size and loop order. Note that the tile size and the loop order together result in the best performance. To the best of our knowledge, there does not exist any work that identifies the best-performing tile size and loop order *together* for nested loops.

In this work, we propose and develop a supervised machine learning model to predict the best-performing (tile size, loop order) combination for a given multi-dimensional loop. More specifically, our machine learning model is a hierarchical classifier based on Support Vector Machine (SVM) [21]. Supervised learning models need representative training data sets for training the model. But research in compilers is often hampered by limited training data sets. To overcome this problem, we use our synthetic loop generator tool to generate synthetic training data of specific characteristics that are representative of real-world loops.

We develop a carefully tuned hierarchical classifier which is trained using synthetic loops generated by our tool and predicts the best-performing tile sizes and loop orders for loops from Polybench [23, 24] test suite. For performance evaluation, we use two different target architectures: (i) an Intel Xeon Cascadelake system [3] with 48 processor cores and (ii) an Intel Xeon Phi Knights Landing (KNL) system [30, 31] with 64 processor cores and 128 Vector Processing Units. The tile size and loop order predicted by our SVM-based classifier results in transformed loops whose performance is within 18% and 9% of the *optimal* performance for two different test data sets on Intel Xeon Cascadelake system and 18% and 13% of the *optimal* performance for two different prefetch configurations on Intel Xeon Phi (KNL) system. Further, our method outperforms state-of-the-art techniques, Pluto [11] and Polly [18] with a geometric mean speedup of 1.35x to 1.58x.

2 BACKGROUND AND MOTIVATION

2.1 Background

Loop transformations transform a given multi-dimensional loop into a form such that the performance of the loop improves while retaining the correctness of its functionality. There are many loop transformation techniques but we will limit ourselves to loop tiling and loop permutation [6]. Loop tiling/blocking transforms a given multi-dimensional loop into a set of loops that perform the exact computation but in a different order. The iteration space of the original loop is divided into tiles/blocks such that when a tile is loaded into the cache, all computations on this tile are completed before moving on to the next tile. The performance of the transformed loop is improved as the resulting loop exhibits better data locality than the original order. Further, tiling strip mines the loop by tile size such that the resulting loop exploits coarse-grain parallelism and fine-grain data-level parallelism if there are no loop-carried data dependencies in the outermost level and the innermost loop levels, respectively. To exploit data locality, determining the size and shape of the tile is important.

Loop interchange transformation enables permuting the loop orders in a nested loop, such that the points in the iteration space are traversed in a different order than the original. Given a nested loop, one or more of its permuted loop orders can be legal (i.e., do not violate data dependencies in the original loop). Each permuted loop order may exploit parallelism and data locality at different loop levels, leading to differing performance. Thus, selecting the best-performing loop order is also important from a compiler optimization viewpoint.

Consider a kernel/loop from `gemver` benchmark of Polybench [23, 24] suite as given in Listing 1. Listing 2 shows the tiled version of this code as generated by Polly [18]. Tiling strip-mines the original 2-dimensional loop into a loop with *four* iterators. The two loop iterators i and j correspond to the inter-tile traversal, while the ii and jj correspond to the intra-tile traversal within the iteration space. The four iterators can be permuted such that the resulting permutation should still satisfy all the dependencies in the original loop. There are 24 such possibilities; however, we consider only 6 of them where the intra-tile iterations happen within the inter-tile iterations. These six loop orders are: $L1(i, j, ii, jj)$, $L2(i, j, jj, ii)$, $L3(i, ii, j, jj)$, $L4(j, i, ii, jj)$, $L5(j, i, jj, ii)$, and $L6(j, jj, i, ii)$.

```
for (i=0;i<N;i++)
  for (j=0;j<N;j++)
    x[i]=x[i]+beta*A[j][i]*y[j];
```

Listing 1: gemver_k2 loop

```
#pragma omp parallel
for (i=0;i<N/T;i++)
  for (j=0;j<N/T;j++)
    for (ii=T*i;ii<T*i+T;ii++)
      for (jj=T*j;jj<T*j+T;jj++)
        x[ii]=x[ii]+beta*A[jj][ii]*y[jj];
```

Listing 2: Tiled gemver_k2 loop

In the tiled version of the code, the tile size is another important parameter that influences the performance of the tiled loop. In our experiments, we consider *six* different tile sizes, i.e., 8, 16, 32, 64, 128, 256. Together there are 36 possible choices (one among *six* tile sizes and independently one of the six orders $L1-L6$) for a given 2-dimensional loop nest. Our model predicts the best-performing tile size (one among *six* tile sizes) and loop order (one among $L1 - L6$) for a given loop nest. As we will discuss later in the paper, the performance (execution times or execution cycles) of these different versions of a loop can differ by one to two orders of magnitude. Hence, selecting the best-performing version is a critical aspect of compiler optimization.

2.2 Motivation

This section presents the motivation for our work using a few example loops taken from the Polybench benchmark suite [23, 24]. We show how the best-performing loop order and tile size change across different problem sizes and prefetch configurations.

2.2.1 Heuristic-based loop order selection can lead to lower performance: Consider a 2-dimensional loop from the *gesummv* benchmark, as shown in Listing 3. This loop has parallelism in the i-dimension i.e., different iterations of the i-loop can be executed in parallel as they are independent of each other. However, the loop is not parallel in the j dimension. Hence, for this loop, only loop orders $L1(i, j, ii, jj)$, $L2(i, j, jj, ii)$, and $L3(i, ii, j, jj)$ exploit coarse-grain i.e., Single Program Multiple Data (SPMD) data-level parallelism at the *outermost* loop. Based on this, Polly [18] and Pluto [11] pick loop order L1.

The normalized execution cycles of *gesummv* loop for two different input sizes ($N = 4096$ and $N = 8192$), two different tile sizes

(8 and 32) and for different loop orders are reported in Table 1. The data in each row is normalized with respect to the *best-performing* loop order (one with the lowest execution time) in that row when executed on Intel Xeon Cascadelake system.[1] Polly and Pluto pick $L1$ as the loop order for this loop based on the heuristic used in the framework. However, the best-performing loop order for input size $N = 4096$ and tile size 8 (row 1 in Table 1) is $L3$ and hence, Polly and Pluto incur a performance loss of 24.3%.

```
for ( i = 0; i <N; i ++)
  for ( j = 0; j <N; j ++)  {
    tmp[ i ]=A[ i ][ j ]* x[ j ]+tmp[ i ];
    y[ i ]=B[ i ][ j ]* x[ j ]+y[ i ];
  }
```

Listing 3: gesummv loop

```
for ( i = 0; i <N; i ++)
  for ( j = 0; j <N; j ++)
    x1[ i ]= x1[ i ]+A[ i ][ j ]* y1[ j ];
```

Listing 4: mvt_k1 loop

Table 1: Performance of *gesummv* benchmark

	Normalized Execution Cycles						Loop Order Identified by	
	L1	L2	L3	L4	L5	L6	Polly	Pluto
4K_TS8	1.243	1.204	**1.000**	3.515	3.626	20.505	L1	L1
8K_TS8	1.168	**1.000**	1.168	3.188	3.579	18.603	L1	L1
8K_TS32	1.261	1.221	**1.000**	1.723	2.041	15.417	L1	L1

One might argue that loop order $L2(i, j, jj, ii)$ can exploit both data-level parallelism at the innermost loop and task-level parallelism at the outermost loop and hence would be a better choice. However, for the given loop and the input size ($N = 4096$), $L2$ incurs a performance loss of 20.4%. The scatter-gather memory accesses (due to `A[ii][jj]` access pattern) required at the innermost loop to exploit the data parallelism may be incurring significant performance overhead.

2.2.2 Best-performing loop order changes across problem sizes: For the same *gesummv* loop shown in Listing 3, the performance for the *six* loop orders for a problem size $N = 8192$ and a tile size 8 is presented in row 2 of Table 1. Here, the best-performing loop order is $L2$. Selecting the loop order $L1$, as given by Polly and Pluto, incurs a performance loss of 16.8%. On the other hand, choosing $L3$ as the loop order (the best-performing loop order for $N = 4096$) also results in a similar performance loss, indicating a trade-off between data parallelism and scatter-gather overhead taking place as the input size changes. Thus, the best-performing loop order can change with problem size, suggesting that the input problem size is an important factor in deciding the loop order.

[1]The details of our experimental framework and methodology are presented in Section 4.

Table 2: Performance of gemver_k1 benchmark

	Normalized Execution Cycles					
	L1	L2	L3	L4	L5	L6
TS_8	1.069	1.063	1.105	2.092	2.266	10.650
TS_16	1.041	1.192	1.142	2.070	5.305	14.385
TS_32	1.716	1.240	1.140	2.308	3.609	18.703
TS_64	**1.000**	3.528	1.189	2.204	4.549	25.496
TS_128	1.695	2.451	1.341	1.407	4.177	16.957
TS_256	2.095	4.967	1.826	1.946	4.644	15.604

2.2.3 Best-performing loop order changes across tile sizes: Next, we consider the tile size 32 and problem size $N = 8192$ for the same *gesummv* loop. The performance for the different loop orders is shown in row 3 of Table 1. Here, $L3$ is the best-performing loop order. Neither Polly nor Pluto identifies $L3$ as the best-performing loop order and hence, incur a performance loss of 26.1%. The $L2$ loop order also incurs a similar performance loss of 22.1%. This shows that best-performing loop order also depends on the tile size. Next, let us look at the impact of tile size on the performance of the transformed loop.

2.2.4 Both tile size and loop order together influence the performance: Let us consider a loop from *gemver*. For a given problem size 4096, we present the performance of the loop, in terms of normalized execution cycles, for different tile sizes and loop orders in Table 2. The values are normalized with respect to the best-performing version (corresponding to tile size = 64 and loop order $L1$). As can be seen from the table, the default tile size (= 32) incurs a performance loss of 14% (for the best-performing loop order) to 18.70 times (for the worst-performing loop order with the default tile size). Further, a wrongly selected tile size and loop order can result in a performance loss that can be as high as 25.50 times. Thus, the compiler transformation should identify the tile size and loop order that results in the lowest execution cycles.

2.2.5 Best-performing tile size changes across problem sizes: Let us consider a loop from the *mvt* benchmark as shown in Listing 4. We present its performance in Table 3 for three different problem sizes, 2048, 4096 and 8192. Each row in this table represents the normalized execution cycles, normalized with respect to the best-performing loop order across all tile sizes for this input size. The first row presents the data for problem size $N = 2048$ and shows that tile size 8 gives the best performance. However, for problem sizes $N = 4096$ and $N = 8192$, tile sizes 16 and 256 result in the best-performing loop versions. This shows that the tile size that results in the best performance changes across problem sizes. Fixing a single tile size for all problem sizes incurs huge performance losses. Loop transformation frameworks like Polly [18] and Pluto [11] either use a default tile size (32) or require the programmer to select the tile size manually. Fixing the tile size to 32 incurs a loss of 80.2% for $N = 2048$, 44.1% for $N = 4096$, and 49.7% for $N = 8192$. Fixing the tile size to 8 incurs a loss in performance as high as 48.9% for $N = 8192$.

2.2.6 Best-performing tile size changes across prefetch configurations: Last, we demonstrate that the hardware prefetch configuration of a processor also influences the selection of best-performing tile size and loop order. For this, we consider two different prefetch

Table 3: Performance of mvt_k1 benchmark

	Normalized Execution Cycles					
	TS_8	**TS_16**	**TS_32**	**TS_64**	**TS_128**	**TS_256**
2K	**1.000**	2.083	1.802	4.741	5.883	1.622
4K	1.018	**1.000**	1.441	1.135	2.947	3.097
8K	1.489	1.500	1.497	1.347	1.134	**1.000**

Table 4: Performance of $atax\_k2$ benchmark

	Normalized Execution Cycles					
	TS_8	**TS_16**	**TS_32**	**TS_64**	**TS_128**	**TS_256**
3K_AE	**1.000**	1.029	1.312	1.623	1.543	3.036
3K_NE	1.254	**1.000**	1.436	1.387	1.867	3.806

configurations, one in which all prefetchers for both $L1$ and $L2$ caches are enabled (referred to as *AllEnable* or *AE* for short) and another in which none of the prefetchers are enabled (referred to as *NoneEnable* or *NE*) on Intel Xeon Phi (KNL) architecture.[2]

The performance of the $atax$ loop for problem size $N = 3072$ is shown in Table 4, rows 1 and 2 for prefetch configurations *AE* and *NE*, respectively. For each tile size, we report the performance of the best-performing loop order (which could be different across different tile sizes), and the values in each row are normalized with respect to the lowest or best-performing tile size. The best-performing tile size for *AE* configuration is 8 while that for *NE* is 16. In the *NE* configuration, using the best-performing tile size (8) for the *AE* configuration results in a performance loss of 25.4%. Similarly, using the default tile size (32) in *AE* and *NE* prefetch configurations incur performance losses of 31.2% and 43.6% respectively.

To summarize, the performance of a loop nest depends on the complex interplay of multiple factors, including the task and data-level parallelism exploited, the abilities of the prefetchers, the synchronization overheads incurred, the data access patterns in the loop, the size (or volume) of the data accessed. This motivates us to formulate this as a machine-learning problem.

3 TILE SIZE AND LOOP ORDER SELECTION PROBLEM

In this work, as mentioned earlier, we consider 2-dimensional perfect loop nests with *six* possible legal permutations. The i or j or both dimensions could be parallel depending on the dependencies present in the loop nest. Depending on the parallelism available in the loop nest, a specific loop order will exploit the coarse-grain data-level parallelism in the respective inter-tile dimension of the tiled code. Fine-grain data-level parallelism is exploited if the corresponding intra-tile dimension happens to be the innermost dimension. We use Polyhedral techniques and the Polly framework [18] to generate tiled code and to identify and exploit the parallelism for the different loop orders.

3.1 Formulation as Machine Learning Problem

As mentioned earlier, for a given loop nest, we consider six different tile sizes and six possible loop orders for each one of them. The problem is to identify the best-performing tile size and loop order combination for the loop nest. We frame this as a classification problem and identify the input features.

3.1.1 Classification Problem: The classification problem we have on hand is a multi-class classification problem. For this, we propose to use the SVM classifier [21]. SVMs work by finding a hyperplane with the highest margin that can distinguish between data of two classes. They work well with both linear and non-linear data. They support linear, polynomial, radial basis function (RBF) and sigmoid kernels [21]. In our work, we propose to use a multi-class SVM classifier that uses both linear and RBF kernels. The classifier is built using training data (loop nests generated using our loop generator tool) whose output classes are already labelled. The features of the training data loops form the input feature vector.

3.1.2 Input Features: The training and test loops are characterized by a few features referred to as input features. In the following discussion, `X` and `A` represent, respectively, one- and two-dimensional arrays and `f` and `g` represent affine functions of loop index variables `i` and `j`. We propose using the following as input features:

(1) The dimension(s) of the original loop nest, i.e., i or j or both i and j, that has (have) parallelism. These are respectively referred to as $pari$, $parj$, and $parb$.
(2) The number of references to 2-dimensional arrays of the form `A[f(i)][g(j)]`.
(3) The number of references to 2-dimensional arrays of the form `A[f(j)][g(i)]`.
(4) The number of references to 1-dimensional arrays of the form `X[f(i)]`.
(5) The number of references to 1-dimensional arrays of the form `X[f(j)]`.
(6) The number of 2-dimensional arrays of the form `A[f(i)][g(j)]`.
(7) The number of 2-dimensional arrays of the form `A[f(j)][g(i)]`.
(8) The number of 1-dimensional arrays of the form `X[f(i)]`.
(9) The number of 1-dimensional arrays of the form `X[f(j)]`.
(10) The problem size[3] N

Different features can use different `f` and `g` functions. These features are carefully selected, using domain knowledge, from a host of features that characterize the performance of the given loop nest for various loop orders and tile sizes.

The chosen input features have bearing on the architecture and impact the performance of the loop nest. The `A[f(i)][g(j)]` vs. `A[f(j)][g(i)]` access patterns along with the loop order capture the locality in accesses, vector loads vs. scatter-gather accesses. Similarly, Parallelism in i- or j-loop indicates parallel outermost loops and vectorizable inner loops for different loop orders. Writes to 1-D arrays and 2-D array access functions capture the dependent loop dimension.

The input features can be easily extracted by a compiler. We refer to the above as feature set 1 or $fs1$ for short. From this, we derive feature set 2 ($fs2$) by retaining features N, the dimensions

[2]We could not disable the prefetch configuration on the Intel Cascadelake system as it is a part of a production cluster.

[3]Our approach and methodology works for any NxM loops although the paper focuses only on NxN loops.

of the original loop nest, i.e., i or j or both i and j, that have parallelism and removing the four features related to the total number of arrays; further four new features related to store operations corresponding to 1- or 2-dimensional arrays and array access patterns are added.

3.2 Hierarchical Classification

One approach to solving the multi-class classification problem is to use *flat* classification, where the output labels belong to one of the 36 classes (corresponding to six different tile sizes and six possible loop orders). While this approach is simple, it fails to adequately capture the interplay between tile size and loop order. Our initial experiments showed that this approach was about 53.09% away from the best-performing one. An alternative approach is to build a hierarchical classifier that predicts either the tile size or the loop order first and then predicts the other parameter. This approach was no better than the flat classification and also fails to capture the interplay between tile size and loop order in the classifier.

We solve the problem by designing a hierarchical classifier based on SVM. We propose two approaches for constructing the hierarchical classifier. One is a systematic approach that explores the design space of hierarchical classifiers. This approach is general and can be applied to any target architecture. The other approach uses domain knowledge and observed characteristics in training loops to come up with a tuned hierarchical classifier. We describe the second approach first.

3.2.1 A Tuned Hierarchical Classifier: After careful study of the training data set and some experimentation on the validation set, we propose a hierarchical classifier with $four$ levels (see Figure. 1). As discussed in Section 2, for loops that are parallel in the i-dimension, loop orders $L1$, $L2$ or $L3$ exploit coarse-grain data-level parallelism at the outermost level. Similarly for j-parallel loops, loop orders $L4$, $L5$ or $L6$ exploit coarse-grain data-level parallelism at the outermost level. Based on this observation, the root node of the tree of classifiers (C1) is designed as a multi-class classifier to classify the loops into one of the two categories of loop orders: $\{\{L1, L2, L3\}, \{L4, L5\}\}$. The classification of loops belonging to $\{L1, L2, L3\}$ into $\{L4, L5\}$ or a vice-versa would incur a large performance loss. We do not consider loop order $L6$, as it infrequently appears as the best-performing loop order in the training data set on Intel Cascadelake architecture. Having obtained a broader loop order category, we then classify the data further based on the tile size at level-2 in the tree of classifiers.

At level-2, the left classifier (C2) predicts the tile size to be one among the group $\{\{T1, T2\}, \{T3, T4\}, \{T5, T6\}\}$, for class $\{L1, L2, L3\}$ loops, as training data set loops that have best-performing tile size of $T1$ (tile size = 8) and $T2$ (tile size = 16) exhibit similar characteristics. A similar observation was also made for the classes $\{T3, T4\}$ and $\{T5, T6\}$. The right classifier (C3) predicts the tile size to be one among $T1 - T6$ for class $\{L4, L5\}$ loops.

At level-3, on the left side, we have three classifiers. The leftmost classifier (C4) classifies the loops for tile sizes $T1$ and $T2$. The middle (C5) and the right (C6) classifiers, respectively, predict the loop order among $L1, L2, L3$ for loops whose predicted tile sizes are $\{T3, T4\}$ and $\{T5, T6\}$. The classifiers on the right side (C7 – C11) are binary classifiers that predict the loop orders among $L4$ and $L5$ for each tile size. At level-4, we have three 3-class classifiers: (C13 –C15) to classify among tile sizes $\{T1, T2\}$, (C16 –C18) to classify among tile sizes $\{T3, T4\}$, (C19 –C21) to classify among tile sizes $\{T5, T6\}$.

The hierarchical classifier obtained for Intel KNL architecture is slightly different from the one shown in Figure 1. We do not present it here due to space constraints.

3.2.2 A Systematic Approach for Classifier Design: Is there a way to systematically design the hierarchical classifier? The design space of the hierarchical classifier is exponential (in the number of tile sizes and loop orders considered). We propose an approach that is effective without incurring exponential search costs.

We start with the root node of the tree of classifiers (C1) to classify the loops into one of the two categories of loop orders: $\{(L1, L2, L3), (L4, L5, L6)\}$, based on the domain knowledge of loop orders of i-parallel and j-parallel loops. Next, for each level, we construct the classifier based either on the tile size or the loop order, depending on whichever one has not yet been fully classified. If the tile size classifier is considered next, then we construct all possible classifiers for that level. For example, if we consider (T1, $\cdots$ T6), then all possible 2-class classifiers (e.g., {(T1, T2, T3), (T4, T5, T6)}, {(T1, T2), (T3, T4, T5, T6)}, etc.), 3-class classifiers (e.g., {(T1, T2), (T3, T4), (T5, T6)}, {(T1, T4), (T2, T5), (T3, T6)}, etc.), 4-class, 5-class, and 6-class classifiers are considered. Although the number of classifiers considered here is large (198), as we will explain later, the design space exploration still does not explode multiplicatively at each level. Among the classifiers considered at this level, we choose the best performing one in terms of the prediction accuracy on the validation set (derived from the training data set) and use that as the classifier at this level. We make the important observation that other classifiers considered at this level need not be explored further as their prediction accuracy is lower, and no matter how good the classifiers at the subsequent levels are, the prediction error made at this level cannot be revoked. Hence, any of those classifiers is *likely* to give lower overall prediction accuracy than the best-performing one at this level.

We continue the above step at each level, choosing one of tile size or loop order classification but exploring all possible classifiers at this level. But as we choose only the best-performing classifier at each level, this essentially ensures that the total number of classifiers explored is the sum of the classifiers explored at each level and not their product. Note that the systematic approach does not guarantee identifying the optimal hierarchical classifier. However, it will potentially find a hierarchical classifier whose performance is close to that of the optimal. Our extensive experimentation shows that our tuned hierarchical classifier (shown in Figure. 1), carefully designed using the domain knowledge, and the one obtained from the systematic approach perform equally well and results in transformed loops whose performance differs only marginal from each other.

3.3 Synthetic Loop Generation

We apply supervised learning method to find the best-performing tile size and loop order for a loop. Supervised learning methods require a large training data set. However, the number of available loops from various benchmark suites such as NAS parallel

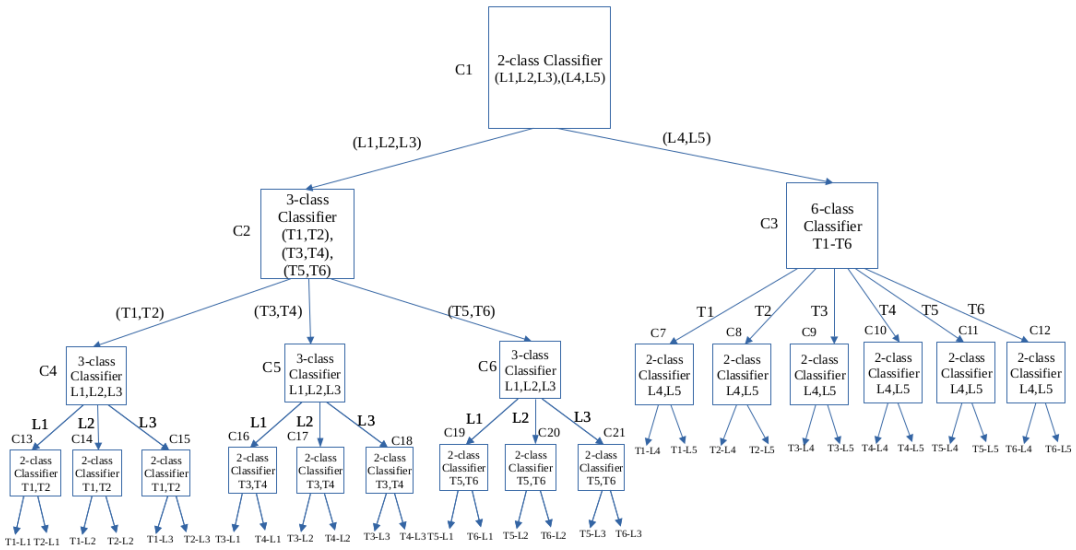

Figure 1: Hierarchical Classifier Design

benchmarks [7, 8], PARSEC [9], and Polybench [23], that are 2-dimensional, tileable and could be used for training are very few, perhaps yielding a few 10's of loops. It is desirable to have a method that can generate thousands or even millions of (synthetic) loops which are different (in terms of several loop characteristics) yet representative of real-world applications. If the method can also generate loops with a given set of loop characteristics, it will also solve the problem of generating targetted loops for a given end goal for which the machine learning technique is applied.

To address this, we have developed a tool to generate synthetic loops automatically [5]. The synthetic loops are characterized by certain properties such as the number of statements, the number and types of data dependencies between the statements, the memory access patterns of one- and two-dimensional arrays, etc. By choosing different values for these parameters, we can generate many loops. In this work, we have considered *perfect*, 2-dimensional[4] loop nests which can be tiled in either dimension. Each loop nest generated is of the form shown in Listing 5, and each statement is in the canonical 3-address form as: `a = b op c`. Each operand in the 3-address statement can be a scalar, 1- or 2-dimensional array with appropriate indices. 2-dimensional arrays can have indices of the form `A[f(i)][g(j)]` or `A[f(j)][g(i)]` where f and g are affine functions. Similarly, 1-dimensional arrays can have an index of the form `A[f(i)]` or `A[f(j)]`. Constraints such as affine accesses and perfect loop-nests are essential to analyze the loops and construct legal transformed versions.

Statements in the loop can have loop-carried or loop-independent dependencies among them. All legal dependencies of the form $(=,=)$, $(=,<)$, $(<,=)$, $(=,*)$, $(*,=)$ are allowed. Further, each dependence can be true, anti-, or output-dependence, and the dependence can be a self-dependence, i.e., from a statement to itself. The dependencies in the loop can be limited to permit parallelism in either or both loop dimensions. We do not consider loops with dependencies in both dimensions, as they do not permit parallelism.

```
for (i=0;i<N;i++)
 for (j=0;j<N;j++)
 {
  S1:A[i][j]=B[j][i]*C[i-1][j+1];
  ...
  Sn: ...
 }
```

Listing 5: Template 2-D Loop Nest

3.3.1 Data Dependence Graph Generator: Each loop can be represented as a directed Data Dependence Graph (DDG) where nodes represent statements of the loop and edges between a pair of nodes represent data dependence between the corresponding statements in the loop. The dependence can be of any of the forms mentioned earlier such that the resulting loop is parallel in at least one of the dimensions. Thus, our method to generate synthetic loops first generates a random DDG and then synthesizes a compilable C-loop corresponding to that DDG.

The tool receives the characteristics of the loop to be generated as an input configuration file. The input file consists of the number of statements (in canonical three-address form) to be present in the loop and the number of dependencies that should be present between these statements. The input file also specifies the types (true, anti-, and output) of dependencies and the dependence distance (in case of true and anti-dependencies) as simple probability values. Further, the configuration file specifies whether additional scalar, 1- or 2- dimensional arrays are used as read-only operands in the loop. Last, the configuration file also specifies whether the

[4]The synthetic loop generation methodology can be extended to higher dimension loop nests as well.

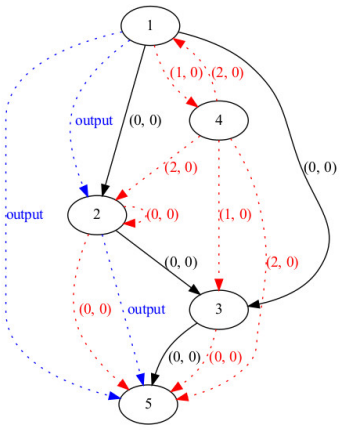

Figure 2: DDG with implied edges

2-dimensional (1-dimensional) array access pattern of the destination and source operands are of the form `A[f(i)][g(j)]` or `A[f(j)][g(i)]` (respectively, `A[f(i)]` or `A[f(j)]`) using probability values. These configuration parameters can be set by observing the respective characteristics of real loops. In our work, we obtained these parameters using a characterization study done on 52 loops of the Polybench-3.2 [23] suite.

Using the values from the input file, our tool generates a random DDG satisfying the above characteristics, involving 1- or 2-dimensional arrays, having parallelism in at least one of the dimensions. The distance vector is assigned randomly in the range (-2,2) for each dimension. We limit the dependence value to this range based on the characteristics observed in real loops. We anticipate that for loops with a dependence distance greater than 2, the distance can be modelled as 2 in the feature vector for predicting tile size and loop order. The transformed loop should, however, use the actual dependence distance to ensure correctness. The nodes of the DDG represent a destination or left-hand side (LHS) of the statement. The pairs of nodes associated with the same destination node will have output dependence. We ensure that our DDG generator generates only *legal DDGs*, i.e., DDGs in which there are no cyclic dependencies among statements that prevent ordering the statements within the loop nest. Our DDG generator tool is implemented in Python and is represented using a dictionary data structure.

3.3.2 Synthesizing C-code for the Generated DDG: . For each node in the DDG, we generate corresponding C-statements in the order of the node number, as node numbering dictates a valid ordering of the statements in the loop. The statements are enclosed within a 2-dimensional nested for-loop. In our training data set, we have constructed programs with up to 35 statements and more than 70 dependence edges. Figure 2 shows the DDG with implied edges added by our tool. The corresponding C-statements synthesized by our tool are shown in Listing 6. We also have implemented a small validation tool using the ISL (Integer Set Library) [35] tool to ensure that our tool generates valid C-statements for a given DDG. The ISL tool takes C-statements generated by our tool and outputs all dependencies between these statements. The DDG constructed from all the dependence relations is then compared with the random DDG generated by our tool.

```
for(i=0;i<N-2;i++) {
 for(j=0;j<N;j++) {
  A[i][j]=C[i+1][j] * a[i];
  A[i][j]=A[i][j] + u0;
  B[i][j]=A[i][j] - b[j];
  C[i][j]=A[i+2][j] * B[i+1][j];
  A[i][j]=B[i][j] * c[i];
 }
}
```

Listing 6: C-code generated by Synthesizer

4 IMPLEMENTATION AND EXPERIMENTAL METHODOLOGY

4.1 Integrating into Compiler Toolchain

We have implemented our tool as a separate module consisting of a machine-learning model which is trained offline. The training cost of the model is an one-time overhead. The target loop should be marked using compiler directives and the loop nest features are extracted and given as input to our model for predicting the best tile size and loop order. The rest of the compiler tool chain can take the output of our model and generate the code. The additional compilation overhead our approach is in 10's of milliseconds, but the benefits are observed in reducing the runtime of the loop nest.

4.2 Architectures Used

We have used two different target architectures, one consisting of a multi-core processor (Intel Xeon Cascadelake 8268 [3]) and the other a many-core processor (Intel Xeon Phi (KNL) [30, 31]) for our experiments. The Intel Cascadelake system used is a part of a production cluster and has 48 cores in 2-socket configuration. It has 192GB RAM (4 GB per core). The base frequency of the processor is 2.9 GHz. The number of threads per core is 2. It has private L1d and L1i caches each of capacity 32KB per core and an L2 cache of capacity 1MB per core. The L3 cache of capacity 35.75MB is shared across all cores in a socket. It supports AVX-512 [12] instructions. Each socket has 2 memory controllers and 3 memory channels per controller. It supports DDR4-2933 memory.

The second system used in our experimental work is based on Intel Xeon Phi (KNL) [30, 31] architecture. The KNL processor used in our work has 32 active tiles, 64 cores, and 128 VPUs. The VPU supports all floating-point computations up to the latest AVX-512 [12] vector instructions, supporting up to eight double-precision floating-point operations per cycle in each VPU. The 16GB direct-mapped multi-channel DRAM (MCDRAM) is in cache mode. It supports L1 cache prefetcher, also known as Instruction Pointer Prefetcher (IPP) and L2 hardware prefetcher.

4.3 Measuring Execution Cycles

We measure the execution cycles for the transformed loop nests using *_rdtsc* function. The code for each of these versions for both the architectures is obtained using the state-of-the-art compiler

Table 5: List of Benchmarks Used

Benchmark Name	No.of loops	Benchmark Name	No.of loops	Benchmark Name	No.of loops
gemver	3	syrk	1	jacobi-2d	1
atax	2	adi	4	gemm	3
gesumm	1	fdtd-apml	2	syr2k	3
mvt	1	fdtd-2d	2		

LLVM 11.0v integrated with Polly tool (opt 11.0) with appropriate flags (polly-parallel, polly-vectorizer=polly, polly-tile-sizes=32,32, mattr=+avx512f, mcpu=cascadelake, -O3). To minimize the variations in execution cycles due to operating environments and to reduce the effect of Operating System related latencies [16, 27], we had exclusive access to the server during the run, and disabled hyperthreading. Further, to obtain robust measurement of execution cycles, we run each program/loop nest 20 times and measure the execution cycles in each execution. From these values, outliers are removed using the standard $1.5 \times InterQuartileRange(IQR)$ [28] method. The resulting values are accepted if the coefficient of variation (CV) is less than or equal to 2% else we repeat the execution ten more times, discard the outliers, and accept the values if the CV is less than or equal to 2%. The average of these (accepted) values is the execution cycle for the loop. The whole run is discarded if the CV is greater than 2% or if we find more than 20% of the outliers are removed. We repeat such execution five times for each program. The final execution cycle value is the average of all five averages taken across five executions. With this rigorous procedure for measuring the execution cycles, the CV was observed to be less than 2% in all our measurements.

4.4 Benchmarks Used

We have used twenty-three 2-dimensional perfect loop nests from Polybench-3.2 [23] benchmark suite that are permutable, tileable in both dimensions and have parallelism in at least one dimension. These loops have one or more statements (up to five statements) but contain complex arithmetic expressions involving multiple operations and operands. Further, we looked at PolyBench-4.2 as well, but could not add any additional kernels to our test set as they were not permutable. The name of the benchmark and the number of loop nests taken from them are listed in Table 5. We refer to this set of loops as Poly_Orig.

To evaluate the performance of our classifier on a different set of test data, we have generated one more set of the same Polybench benchmarks, where each statement of the loop is represented in the canonical 3-address form (similar to Listing 5 using temporary arrays). A loop of Polybench in its original form and its 3-address form performs the same operations and produces the same output; only the representation has changed. The canonical form also changes the features of the loop nest and the introduction of temporary arrays may influence the best-performing loop order due to increased memory footprint/working set and the associated cache behavior. We refer to this set of loops as Poly_3-AddressCode or Poly_3AC for short. We have used five different N (problem size) values, namely $N = 2048$, $N = 3072$, $N = 4096$, $N = 6144$, and $N = 8192$, in our evaluation. We identify the best-performing tile size and loop order for each loop nest by running these six different loop orders for all six tile sizes on the Intel Xeon Cascadelake and Intel KNL system and selecting the one with the lowest execution cycles as its output class.

4.5 Support Vector Machine Classifier

As explained in Section 3.2, our classifier is developed using Support Vector Machines (SVM)[13] to predict the best-performing tile size and loop order. We have experimented with different classification methods including decision tree classifiers such as C5.0 and random forest, as well as Linear Regression models for predicting the execution time of different loop orders. We chose SVM as our base method and the hierarchical classifier, as their prediction resulted in better performance.

We have used the SVM implementation `e1071` [21], which solves a multi-class classification problem using a one-against-one approach by constructing $n(n-1)/2$ binary classifiers (where n denotes the number of classes) and the class label of a data point is found using a majority vote.

The training and test data were normalized to zero mean and unit variance for all our experiments and results discussed in the next section. We have explored both linear and radial basis function (RBF) kernels of the `e1071` package of R(version 3.6.0) for our hierarchical classification. Every classifier design in the hierarchical classification approach requires extensive experimentation involving cross-validation techniques. Ten percent of the training data was set aside as a validation set in our experiments. For every classifier in Figure 1, the kernel function (linear/RBF), the corresponding hyperparameters, and the feature set were chosen based on the performance of the validation set.

5 RESULTS AND DISCUSSIONS

This section discusses the results of our experiments on Intel Cascadelake [3] and Intel Xeon Phi (KNL) [30, 31] systems. On KNL, we consider two hardware prefetch configurations: one where all prefetchers are enabled (All_Enable or AE for short) and another where none of the prefetchers are enabled (None_Enable or NE). As the Cascadelake system used is a part of a production cluster, we could only explore the AE prefetch configuration.

We refer to our tuned hierarchical classifier (shown in Figure 1) as *HC_Tuned*, and compare its execution cycles with an oracle method (referred to as *Opt*) that always picks the best-performing tile size and loop order, among the 36 possible (tile size, loop order) combinations. This is a "lower is better metric" and indicates how far the performance of the identified tile size and loop order of the method is from the *best-performing* one. Further, when reporting the performance for a set of test loops, we take the geometric mean of the normalized execution cycles across different loops in the test set. Last, we also show the performance of the hierarchical classifier obtained using the systematic approach, referred to as *HC_SysApp*, and compare its performance with *HC_Tuned*.

We compare the performance of *HC_Tuned* and *HC_SysApp* with (i) an oracle method that always picks the best-performing loop order for a given tile size 32, lowest execution cycles among the six loop orders, referred to as *T32_Opt*; (ii) an SVM based loop order predictor [5], referred to as *T32_Pred*, for the tile size 32; (iii) the code generated by Polly [18] and Pluto [11] for tile sizes 32

Table 6: Performance using Synthetic Benchmarks as Training Data on Intel Cascadelake System

Test Set	Pref. Config.	Geometric Mean of Normalized Execution Cycles									
		HC_Tuned	HC_SysApp	T32_Opt	T32_Pred	Polly_32	Pluto_32	Polly_64	Pluto_64	Polly_Opt	Pluto_Opt
Poly_Orig	AE	1.18	1.22	1.31	1.57	1.84	1.64	2.02	1.78	1.33	1.15
Poly_3AC	AE	1.09	1.16	1.21	1.31	1.72	1.66	1.80	1.66	1.42	1.29

Table 7: Performance using Synthetic Benchmarks as Training Data on Intel Xeon Phi (KNL) System

Test Set	Pref. Config.	Geometric Mean of Normalized Execution Cycles									
		HC_Tuned	HC_SysApp	T32_Opt	T32_Pred	Polly_32	Pluto_32	Polly_64	Pluto_64	Polly_Opt	Pluto_Opt
Poly_Orig	AE	1.18	1.17	1.41	1.84	2.64	2.18	3.63	2.58	1.88	1.36
Poly_3AC	AE	1.07	1.07	1.33	1.50	2.59	2.08	2.93	2.18	1.98	1.43
Poly_Orig	NE	1.13	1.14	1.39	1.65	2.20	1.87	3.13	2.35	1.53	1.22
Poly_3AC	NE	1.17	1.14	1.27	1.45	1.89	1.56	2.26	1.74	1.48	1.15

and 64; and (iv) *Polly_Opt* and *Pluto_Opt* methods which refer to, respectively, the code generated by Polly [18] and Pluto [11] for the tile size that results in best performance. Note that all methods suffixed as *Opt* are oracle in nature. We do not compare with PPCG [36] for C target as it does not support vectorization.

5.1 Performance Comparison

Table 6 reports the geometric mean of normalized execution cycles on Intel Cascadelake architecture for different methods, normalized w.r.t. the oracle *Opt* method. In Table 7 we report the performance numbers for Intel KNL architecture. On both architectures, both *HC_Tuned* and *HC_SysApp* perform consistently better across all test loop sets and applicable prefetch configurations. The performance of *HC_Tuned* differs from that of *HC_SysApp* only marginally. Henceforth, we compare the performance of other methods with one of them (*HC_Tuned*), while the observations hold equally well for the other (*HC_SysApp*).

On the Intel Cascadelake system, both *HC_Tuned* and *HC_SysApp* perform consistently better across both the test sets, achieving the performance of 18% and 9% from the *optimal performance* that can only be achieved by an *oracle method*. T32_Opt incurs higher performance losses of 31% and 21% (i.e., has normalized execution cycles of 1.31 and 1.21), while T32_Pred incurs even higher performance losses of 57% and 31%. The higher performance loss value is due to selecting sub-optimal loop orders and tile sizes. *HC_Tuned* achieves a performance improvement of 1.11x for both Poly_Orig (1.31/1.18) and Poly_3AC(1.21/1.09) over T32_Opt and 1.33x (=1.57/1.18) and 1.20x (=1.31/1.09) for Poly_Orig and Poly_3AC, respectively, compared to T32_Pred. [5]

HC_Tuned outperforms Polly and Pluto for fixed tile sizes (32 and 64) by a significant margin (1.39x to 1.71x). Also, *HC_Tuned* performs significantly better than Polly_Opt and Pluto_Opt (where the chosen tile size is the best performing one for the given loop) in all but one case. Note that Pluto_Opt and Polly_Opt require an *oracle* predictor for the tile size and are not practical. The performance of *HC_Tuned* is 1.35x and 1.42x better than Pluto [11] for Poly_Orig and Poly_3AC, respectively, when the tile size is fixed at 32. Similarly, *HC_Tuned* is 1.55x and 1.58x faster than Polly [18] for Poly_Orig and Poly_3AC, for the default tile size 32.

HC_Tuned predicts the correct loop order for about 60% and 73% loops, respectively, for Poly_Orig and Poly_3AC test sets. Also, we observe that when *HC_Tuned* makes incorrect loop order predictions, the predictions are made within the $L1, L2, L3$ or $L4, L5, L6$ groups, depending on the outermost parallel dimension. Hence, even if the tile size is incorrectly predicted, predicting the correct loop order or one among the two groups of loop orders reduces performance loss. While this is true for Polly and Pluto too, fixing the tile size across different loops and problem sizes is the cause of the significant performance loss. The tile size 32 is best-performing only in about 7% to 11% of the loops. Further, Polly does not take data parallelism due to vectorization into account wherever possible. Pluto accounts for vectorization; however, it does not consider problem size and the interplay between multiple architectural features for selecting the loop orders. Thus, on average, *HC_Tuned* is 1.5x faster than Pluto and Polly.

Even on Intel KNL system, *HC_Tuned* and *HC_SysApp* perform significantly better than all other methods for both the test sets and prefetch configurations. The performance improvement over other competitive methods exhibit a similar trend to that seen for the Cascadelake system. Further, *HC_Tuned* and *HC_SysApp* perform considerably well even under the NE prefetch configuration on the KNL system again for all competitive methods. Only for Poly_3AC loops with NE configuration, Pluto_Opt performs marginally better than *HC_Tuned* by 2%; however, as stated before, it is an oracle predictor and not practical. The performance of *HC_Tuned* for tile size 32 is 1.85x and 1.66x better than Pluto [11] and 2.24x and 1.95x better than Polly [18] for AE and NE configurations, respectively.

5.2 Where does *HC_Tuned* lose its Performance?

Compared to *Opt*, *HC_Tuned* results in 9% to 18% performance loss on the Intel Cascadelake system. Where does this loss come from? To understand this, we present a confusion matrix-like structure in Figure. 3, that reports the performance loss under each case of misprediction. We show the performance loss of our *HC_Tuned* model (against Opt) for each (tile size, loop order) combination of true (in rows) and predicted classes (in columns) for the Poly_Orig test set. The confusion matrix has 36×36 cells corresponding to six

[5] When comparing two competitive methods, e.g., HC_Tuned and T32_Pred, the performance improvement is obtained by taking the ratio of normalized execution times of T32_Pred and HC_Tuned.

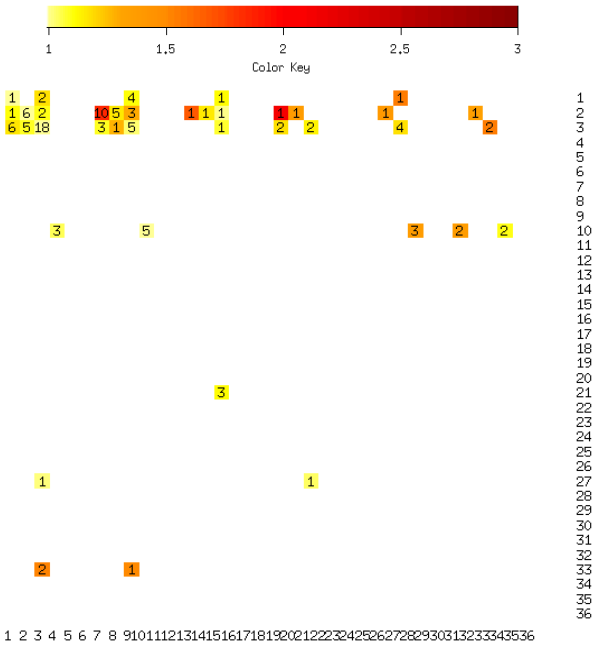

Figure 3: HC_Tuned Performance for AE Poly_Orig Test Set

tile sizes and six loop orders. Many cells in the confusion matrix are empty (does not have any loops that fall under this category). The number in each cell represents the number of test loops for that combination. The color coding of each cell indicates the relative performance, obtained as the geometric mean of normalized execution cycles for all those test loops, w.r.t. the best-performing one. Diagonal cells, if present, will have a performance of 1, as the predicted combination is indeed the best-performing one.

First, from Figure. 3, we observe that *only* a few cells have a higher intensity (orange or red colored) while others (yellow colored) have moderate performance loss (less than 1.25x). More specifically, there are seven cells with geometric means in the range 1.25–1.4 and six cells in the range 1.5–1.85. We observe that for seven out of these thirteen cells, the loop orders are predicted correctly for loops in these regions, but the tile size is sub-optimal. For the remaining six cells, both tile size and loop orders are sub-optimal. When a cell in the confusion matrix consists of multiple loops (e.g., 6 in one of the dark yellow cells), not all contribute to worse performance. The value reported in the cell is the geometric mean; therefore, some of the loops in that cell could still have a lower performance loss. For cells with a performance loss in the range 1.25–1.4, about 30% – 50% of the loops contribute to higher performance loss, while for cells with performance loss in the range 1.5–1.7, 40% – 50% of the loops have higher performance loss. Further, one cell with a true value of 7 and a predicted value of 2 has a count of ten loops with a geometric mean of 1.84. Here one loop is mispredicted across all five problem sizes and another loop is mispredicted for three problem sizes. For them, a sub-optimal loop order is picked, which incurs scatter-gather memory accesses, thus significantly adding to the performance loss. These are the regions that require further fine-tuning of the model.

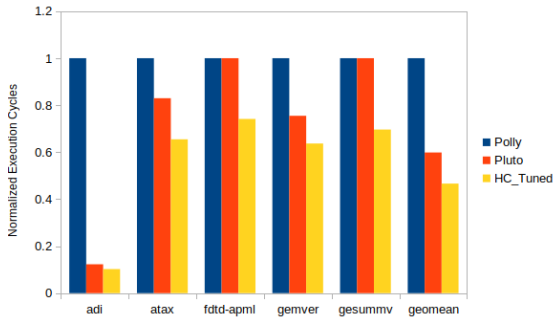

Figure 4: Full Application Performance with *HC_Tuned*

5.3 Full Application Performance

To show the benefits of using our approach for the entire application, we have chosen five applications from the Polybench-3.2 [23, 24] suite, with more than one kernel. We consider three problem sizes i.e., 2048, 4096 and 8192 and measure the execution cycles taken for the entire application, by applying the tile size and the loop order given by *HC_Tuned* for each kernel and compare with Polly [18] and Pluto [11]. For Polly [18] and Pluto [11], the default tile size of 32 along with the loop orders given by them are applied for each kernel of the application. Figure 4 shows the normalized execution cycles normalized with respect to Polly for problem size 8192. We do not present the performance results for 2048 and 4096, as they show similar trends. *HC_Tuned* takes the lowest execution cycles as compared to Polly and Pluto for all five applications. Specifically, *HC_Tuned* achieves a performance improvement of nearly 50% and 18.50% over Polly and Pluto, respectively, for these applications across all three problem sizes. The improvement in performance of each kernel contributes to the overall improvement of the entire application.

5.4 Extending to Higher Dimension Loops

Our approach of building a machine learning based SVM model and the synthetic loop generator are not limited by loop dimensionality. They can be easily extended to handle higher-dimension loops. However, as the loop dimension increases, the number of loop orders to explore increases significantly. For a 3-D loop, there are 90 loop orders to be explored for each tile size, which will require a larger training data set to handle all possible loop orders and thus, could be computationally more expensive and challenging to classify accurately. To reduce the complexities associated with higher dimensional loop nests, we use a simple approach where our technique is applied only to the innermost two loop orders. To decide the innermost loops we use a simple heuristic based on the parallel dimension, number of 2-D arrays vectorized or scattered/gathered, number of write-optimizations, and spatial locality exploited by the loop permutation. On the selected loop permutation, the innermost two loop orders are tiled and our approach is applied on them. We have tested this approach, for a problem size of 2048, on five 3-D test loops consisting of two loops from the Polybench [23, 24] suite and three hand-generated loops, to include different array access patterns. The geometric mean of normalized execution cycles of the loop orders identified by *HC_Tuned* is within 21.29% of the best performing loop order among the 108

i.e., 36 loop orders for each of the three loop permutations. This demonstrates that our approach can be extended to higher dimensional loops.

5.5 How Many Models do We Require?

For Intel Xeon Phi (KNL) system, the hierarchical classifier used for the AE configuration is slightly different from the hierarchical classifier used for the NE configuration. What would happen if we use the same hierarchical classifier for both configurations? We used the AE model for prediction in NE configuration (and vice-versa) and found that the additional performance loss is only marginal (1% and 4%, respectively), and they performed better than all competing models (T32_Opt to Pluto_Opt in Table 7).

5.6 Unknown Input Sizes

To handle cases where the input size value, N is not known at compile time, we predicted the (loop-order, tile-size) variant for one input size (4096) and measured the performance loss of that variant on other input sizes for different loopnests. The additional performance loss varied between -4% to 2%, on an average, with marginal improvement or decrease in the performance loss. This indicates that when the input sizes are not known, one could use an arbitrary size, and additional performance loss is still within tolerable limits. Further such predictions (from a fixed input size) performed significantly better than Polly_32 or Pluto_32.

6 RELATED WORK

Girbal et al. [17] present a framework based on unified representation of loops and statements to support program transformations such as loop fusion, tiling, array forward substitution, etc., and compositions of these transformations. Trifunovic et al. [34] present a fast and accurate cost model and a framework to extract vectorization opportunities using polyhedral representation. Pouchet et al. [26] propose the decomposition of the optimization problem represented as convex polyhedron into subproblems of much lower complexity, introducing *fusibility* concept in PoCC [25]. Bondhugula et al. [10] develop a framework for automatic parallelization and data locality optimization of imperfectly nested loops in the polyhedral model to minimize inter-tile communication volume. A framework for integrated data locality, multi-core parallelism, and SIMD execution of programs was proposed in [20] using *codelets*. However, these approaches always generate the same transformed code for a loop nest, irrespective of the problem size or prefetch configuration or tile size. As our work demonstrates, these approaches, hence may incur significant performance losses.

An end-to-end, fully automatic framework driven by an integer linear optimization framework that finds out good ways of tiling for parallelism and locality using affine transformations is proposed in [11]. Grosser et al.[18] implement polyhedral techniques on top of the LLVM framework to transform and optimize parts of the program in a language-independent way. While [18] attempts to exploit task-level parallelism and data locality; it does not target data-level parallelism/SIMD vectorization. We have performed a quantitative comparison of our method with [11] and [18]. The work reported in [37] and [22] propose analytical models for identifying tile size and and target to optimize any one cache hierarchy level. They do not consider all possible loop permutations i.e., [22] favors vectorizable innermost loops. They consider rectangular and parallelogram tiles of arbitrary size. We currently focus only on square and power-of-2 tile sizes, which in general, perform well from locality, prefetching, and vectorization point of view as cacheline size, cache size, and vector width are all power-of-2.

Stock et al. [33] develop an ML model trained using tensor contractions (TCs) for loop permutation, vectorized loop, and unroll-and-jam optimizations. In [1], the authors develop an ML model for iterative optimization for simple transformations like loop unrolling, common subexpression elimination, etc. Ashouri et al. [4] apply ML techniques to predict the phase order of compiler optimization sequences and models the speedup predictor that selects the best set of compiler optimization sequences. Haj-Ali et al. [19] use different machine learning methods to come up with an auto-vectorization method, but focussing only on vectorization. Stephenson et al. [32] apply supervised classification to predict the unroll factors for loops. A framework for synthesizing C programs using a combination of web crawling and type inferencing by filling the missing pieces is proposed in [15]. Cummins et al. [14] propose CLgen to mine open-source repositories for OpenCL program fragments and apply deep learning techniques to construct models for generating programs automatically. The methodologies do not generate nested loops with dependence characteristics akin to loop nest in real-world programs. OpenTuner [2] ensembles different search techniques for autotuning a code for a given architecture based on the architectural parameters. Sioutas et al.[29] developed an analytical model targetting Halide DSL that selects the cache hierarchy level to optimize and the tile size such that cache misses are reduced. These focus more on memory hierarchy and domain-specific languages.

7 CONCLUSION

In this work, we proposed a technique for identifying the best-performing tile size and loop order for a given loop nest using a supervised machine-learning approach. Our approach builds a Support Vector Machine (SVM) based hierarchical classifier. Our proposed model identifies best-performing tile sizes and loop orders for Intel Xeon Cascadelake system, which are within 18% and 9% of the *optimal* performance for two test data sets and outperforms state-of-the-art techniques, Pluto and Polly. The proposed approach is generalizable to other architectures with the similar architecture characteristics.

ACKNOWLEDGMENTS

We would like to acknowledge the research fundings received from Intel Technology India Private Limited and Volvo Group India Private Limited which supported this work. We are grateful to Prof. Ramakrishna Upadrasta, Indian Institute of Technology, Hyderabad, for the fruitful discussions and his insightful suggestions. We thank Ms. Prashanthi S K for her help in the initial parts of the synthetic loop generator tool.

REFERENCES

[1] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 295–305.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[3] Mohamed Arafa, Bahaa Fahim, Sailesh Kottapalli, Akhilesh Kumar, Lily P Looi, Sreenivas Mandava, Andy Rudoff, Ian M Steiner, Bob Valentine, Geetha Vedaraman, et al. 2019. Cascade lake: Next generation intel xeon scalable processor. *IEEE Micro* 39, 2 (2019), 29–36.

[4] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–28.

[5] Shilpa Babalad, Shirish K Shevade, Matthew Jacob Thazhuthaveetil, and R Govindarajan. 2023. A Machine Learning Approach to Identify the Best-Performing Loop Order. https://github.com/knightlander2023/OptLoopOrder, Technical Report, Department of Computer Science and Automation, Indian Institute of Science, Bengaluru.

[6] David F Bacon, Susan L Graham, and Oliver J Sharp. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)* 26, 4 (1994), 345–420.

[7] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. 1995. *The NAS parallel benchmarks 2.0.* Technical Report. Technical Report NAS-95-020, NASA Ames Research Center.

[8] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 158–165.

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

[10] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. 2008. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*. Springer, 132–146.

[11] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 101–113.

[12] Marius Cornea. 2015. Intel AVX-512 instructions and their use in the implementation of math functions. *Intel Corporation* (2015), 1–20.

[13] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[14] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 86–99.

[15] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando Magno Quinão Pereira. 2021. AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 378–390.

[16] Pradipta De, Ravi Kothari, and Vijay Mann. 2007. Identifying sources of operating system jitter through fine-grained kernel instrumentation. In *2007 IEEE International Conference on Cluster Computing*. IEEE, 331–340.

[17] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317.

[18] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

[19] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[20] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 127–138.

[21] David Meyer, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, Friedrich Leisch, Chih-Chung Chang, Chih-Chen Lin, and Maintainer David Meyer. 2019. Package 'e1071'. *The R Journal* (2019).

[22] Kumudha Narasimhan, Aravind Acharya, Abhinav Baid, and Uday Bondhugula. 2021. A practical tile size selection model for affine loop nests. In *Proceedings of the ACM International Conference on Supercomputing*. 27–39.

[23] LN Pouchet. 2012. Polybench: The polyhedral benchmark suite. http://www.cs.ucla.edu/pouchet/software/polybench.

[24] LN Pouchet and Scott Grauer-Gray. 2011. PolyBench: The Polyhedral Benchmark suite (2011), Version 3.2. http://www-roc.inria.fr/~pouchet/software/polybench.

[25] Louis-Noël Pouchet, C. Bastoul, and U. Bondhugula. 2019. PoCC: the polyhedral compiler collection. http://web.cs.ucla.edu/~pouchet/software/pocc/.

[26] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, Ponnuswamy Sadayappan, and Nicolas Vasilache. 2011. Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices* 46, 1 (2011), 549–562.

[27] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. 2012. Thread tranquilizer: Dynamically reducing performance variation. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 1–21.

[28] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.

[29] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. 2018. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 254–264.

[30] Avinash Sodani. 2015. Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–24.

[31] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-generation Intel Xeon Phi product. *IEEE Micro* 36, 2 (2016), 34–46.

[32] Mark Stephenson and Saman Amarasinghe. 2005. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*. IEEE, 123–134.

[33] Kevin Stock, Louis-Noël Pouchet, and P Sadayappan. 2012. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 1–23.

[34] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 327–337.

[35] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.

[36] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[37] Rui Xu, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Yuhong Song, and Han Wang. 2023. Loop interchange and tiling for multi-dimensional loops to minimize write operations on NVMs. *Journal of Systems Architecture* 135 (2023), 102799.