Identifying Optimal Loop Order for Many-Core Architectures Using A Machine Learning Approach

Abstract: Loop transformation techniques like loop tiling and loop interchange are employed in multi-dimensional loops to expose task and data-level parallelism as well as to improve data locality and reuse. The effect of these loop transformation techniques in many-core architecture is rather complex due to the interplay of data reuse in multi-level cache hierarchy, a variety of prefetching techniques employed by the architecture, and the ability to exploit both task and data-level parallelism. Thus, identifying the optimal loop order, i.e., the loop permutation that achieves the lowest execution time, is important in many-core architectures. However, as our study on loops from the Polybench suite reveals, state-of-the-art loop transformation methods often identify sub-optimal loop orders, which result in significant performance loss.

In this work, we propose using a supervised learning technique for identifying the optimal loop order for a given loop nest. However, supervised learning requires a large body of training data and training data that captures the characteristics of real-world loops. Unfortunately, the number of real-world loops is limited, and hence, the application of supervised learning techniques in compilers is often handicapped with small/insufficient training data. In this paper, we develop a tool to generate synthetic loops with specifiable characteristics that we use to address the problem of inadequate training sets. We develop a Support Vector Machine (SVM) based classifier, trained using the data set generated by our tool, to identify the optimal loop order. We then study the use of an ensemble model to make the prediction of optimal loop order robust. We demonstrate that our approach results in identifying loop orders whose performance is within 4.23% (for all enabled prefetch configuration of the many-core processor) and 8.74% (for none enabled prefetch configuration) of the *optimal* loop order. Further, our method outperforms state-of-the-art techniques, Pluto and Polly, by 50.02%, 9.71%, and 93.78%, 44.79%, respectively.

Keywords: Compiling for Parallelization/Vectorization, loop transformations, machine learning, SVM, Synthetic loop generation

1 Introduction

Many real-world applications contain multi-dimensional loops, and a significant amount of their execution time is spent in executing these loops. The performance of these loops on modern multi-/many-core processors is impacted by the effective memory bandwidth delivered to the loops and the parallelism that can be exploited. There are two different types of data-level parallelism, namely coarse-grain and fine-grain, that are exploited by the multi-core processors. Fine-grain data-level parallelism is typically exploited by the SIMD units, using vector instruction sets, such as SSE(Lomont, 2011; Amiri and Shahbahrami, 2020) or AVX(Lomont, 2011; Cornea, 2015). Hence, the fine-grain parallelism is also referred to as SIMD parallelism. On the other hand, coarse-grain (task-level or data-level) parallelism is exploited by multiple independent cores in the processor. If this parallelism is extracted using domain-decomposition methods, where the same computation is performed on different sets of data, using Single Program Multiple Data (SPMD), we refer to it as coarse-grain data-level parallelism or as SPMD parallelism. Several loop transformation techniques like loop tiling, loop permutation (interchange), etc. are applied on the loops to improve their performance on modern multi-/many-core architectures (Burger, 2005; Gepner and Kowalik, 2006; Held et al., 2006). These techniques enhance the performance of a loop by transforming the original loop order into one that improves its data locality and reuse, and exposes data-level parallelism that can exploit multiple cores and the vector processing units available, if any.

Applying just the two loop transformations i.e., loop tiling (Wolfe, 1987; Xue, 2000) and loop interchange (Allen and Kennedy, 1984; Yi and Kennedy, 2004) for a given multi-dimensional loop alone can result in multiple valid transformed loop orders with different tiling parameters. The performance of a transformed loop on a many-core architecture further depends on a number of architectural features like multi-level cache hierarchy and hardware prefetch configurations. The loop transformed by conventional loop transformation techniques may not realize optimal performance due to the above-mentioned complexities of many-core architectures and their interplay. As such, identifying the loop order that results in the lowest execution time (henceforth called the *optimal loop order* for the loop nest) for a given multi-/ many-core architecture is complex.

Preliminary analysis of applying state-of-the-art polyhedral loop transformation techniques (Grosser

et al., 2011; Bondhugula et al., 2008b) on simple 2-dimensional nested loops, even with a fixed tiling factor, do not always result in identifying the optimal loop order for the loop for the Intel Xeon Phi Knights Landing (KNL) many-core architecture (Sodani, 2015; Sodani et al., 2016). The performance loss incurred by such suboptimal loop orders can be as high as 1.5x-2x over optimal loop order. We note here that Polyhedral techniques are indeed quite useful in identifying and exposing different types of parallelism, data locality, and reuse and are essential for transforming the given loop nest into a legal loop order (which obeys all the data dependencies present in the original loop). However, they fail in picking the best-performing loop order among the many legal orders. The former is often influenced by multiple, complex architecture features. Thus, there is a critical need for techniques to identify the optimal loop order for many-core architectures which work on top of existing Polyhedral loop transformations.

In this work, we propose a supervised machine learning technique to identify the optimal loop order for a given loop. Supervised learning involves building a function that maps input features to target prediction output class/value based on a training data set. In particular, we propose to build a simple classifier for identifying the best performing loop order. To train the classifier, there is a need for a large amount of representative real-world loops that can be used as training data sets. However, the number of real-world loops available as benchmark loops is limited (from a few 10s to maybe a few 100s). Hence the application of the supervised learning methods in compilers is often hampered by the limited training data sets.

To overcome this problem of limited training data set, in this paper we develop a synthetic loop generator tool. The synthetic loops generated by our tool are modelled based on real-world loops. However, their characteristics, such as the dimensionality of the loop, number of statements in the loop, type of data dependencies, the (dependence) distance vectors, and the data access patterns (modeling the locality and reuse distances) are parameterized. Some of these properties are also used as (input) features for the classifier we are interested in. By varying these parameterized values, we can generate several millions of synthetic loops which are representative of real-world loops. We show that these loops can be used as training data sets in building classifiers which can be used/tested on benchmark/real-word loops to identify optimal loop order. Further, by refining appropriate parameters, our proposed synthetic generation method has the ability to generate loops with the required characteristics.

Using the training set generated by our synthetic benchmark generator tool, we train and build a Support Vector Machine (SVM) (Meyer and Wien, 2015; Meyer et al., 2019) based classifier to predict the optimal loop order for the test data set. The classifier is specifically built for identifying the optimal loop order for Intel Xeon Phi Knights Landing architecture which can exploit coarse- and fine-grain data-level parallelism, respectively, on multiple cores and the vector processing units. Besides, we explore two different data prefetch configurations of the architecture (one in which all available hardware prefetchers are enabled and another in which none of the prefetchers are enabled). We also train and build an ensemble model of SVM classifiers where each classifier uses a subset of the train data set generated by our tool. The ensemble model ensures high and robust prediction accuracy. Our ensemble model with nine or more classifiers and 900 or more train data points identifies near-optimal loop orders which are within 4.23% and 8.74% of the optimal loop order for the two prefetch configurations studied. We establish the generalization ability of the ensemble model using synthetic loops generated as the test set. Further, our approach performs better than Pluto (Bondhugula et al., 2008b) and Polly (Grosser et al., 2011), two state-of-the-art open-source compiler frameworks.

1.1 Our Contributions

Our key contributions in this work are:

- We develop a tool to generate synthetic loops with certain properties that can be used as training data set in supervised learning techniques.
- We propose an SVM (Meyer and Wien, 2015; Meyer et al., 2019) based classifier, which is trained using the train data set generated by our tool to predict optimal loop orders for the test data set for Intel Xeon Phi KNL (Sodani, 2015; Sodani et al., 2016) architecture for two different prefetch configurations.
- We build an ensemble model of classifiers with SVM (Meyer and Wien, 2015; Meyer et al., 2019) which uses subsets of train data set generated by our tool. Our ensemble model identifies loop orders which perform, on average, within 4.23% and 8.74% of the optimal loop order for two prefetch configurations.
- We also establish the generalization ability of our model to predict the optimal loop orders for any test data set by using the synthetic loops generated as the test set.
- Compared to Pluto (Bondhugula et al., 2008b) and Polly (Grosser et al., 2011), two state-of-the-art open-source compiler frameworks, the loop orders identified by our classifier achieve a performance improvement of 50.02% and 93.78%, respectively, for prefetchers enabled configuration and 9.71% and 44.79% for prefetchers disabled configuration.

The paper is organized as follows. Section 2 discusses the necessary background and provides the motivation for our work. Section 3 presents our approach of identifying optimal loop order as a classification problem

and generation of synthetic loops. Section 4 reports our experimental methodology. In Section 5, we present our results. In Section 6, we discuss related work. Section 7 provides concluding remarks.

2 Motivation

In Section 2.1, first, we will provide the necessary background on loop transformation and Intel Xeon Phi Knights Landing architecture. Subsequently, in Section 2.2, we motivate our work with the help of a few examples.

2.1 Background

2.1.1 Loop Transformation

In this paper, we will limit ourselves to two loop transformations, namely loop tiling and loop permutation (interchange). Loop interchange transformation permutes the loop order in a nested loop. For example, in a two-dimensional loop nest, loop interchange corresponds to changing the outer loop as inner loop and inner loop as the outer loop. Loop interchange essentially walks the points in the iteration space in an order different from the original order. The tiling transformation divides the iteration space of a loop into smaller blocks called tiles, such that when a tile of data is loaded into memory, all the computations associated with that data are completed before moving on to the next tile. This enhances exploiting data reuse. Further, tiling stripmines the loop by tile size, which enables exploiting fine-grain data-level parallelism if the innermost loop is data parallel. In this paper, we will limit ourselves to a fixed tile size T. Both loop tiling and loop interchange transformations are legal only if the transformed loop satisfies all the data dependencies present in the original loop.

Consider a 2-dimensional loop from `gemver` program, shown in Listing 1, from the Polybench benchmark suite (Pouchet, 2012; Pouchet and Grauer-Gray, 2011).

```
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_k2loop

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. Specifically, the value produced for `x[i]` in each iteration of the innermost loop depends on the value of `x[i]` written in the previous j iteration. Hence, different j-iteration cannot be executed in parallel. Next, we will consider a tiled version of the loop, assuming N to be a multiple of T. The tiled version of the loop is as shown in Listing 2. As the parallel i-dimension is in the outermost loop, it exploits coarse-grain data-level parallelism at the highest level of granularity on multiple cores. As the conditions for loop interchange hold (on the statement(s) in the loop) (Allen and Kennedy, 1984; Yi and Kennedy, 2004), all the four loop iterators i, j, ii, jj can be permuted or interchanged to get 24 different loop orders; however, we will consider only six different orders in which the intra-tile iterations happen inside the inter-tile iterations.

In this paper, we restrict ourselves to 2-dimension perfect loop nests, which allow interchange of i and j dimensions. Hence the tiled version of the loop-nest will always permit all the 6 permutations of the loop nest. Let us denote the six valid loop orders along with the loop iterators as: $L1(i,j,ii,jj)$, $L2(i,j,jj,ii)$, $L3(i,ii,j,jj)$, $L4(j,i,ii,jj)$, $L5(j,i,jj,ii)$, and $L6(j,jj,i,ii)$. Among these loops order, for the above loop, only loop orders $L1(i,j,ii,jj)$, $L2(i,j,jj,ii)$ and $L3(i,ii,j,jj)$ exploit coarse-grain (SPMD) data-level parallelism at the outermost loop. For loop orders $L4(j,i,ii,jj)$, $L5(j,i,jj,ii)$, and $L6(j,jj,i,ii)$ the parallel threads (at the i-iterator) need to synchronize for each j iteration. Hence polyhedral frameworks such as Pluto (Bondhugula et al., 2008b) and Polly (Grosser et al., 2011) prefer the loop order $L1(i,j,ii,jj)$, $L2(i,j,jj,ii)$ and $L3(i,ii,j,jj)$ for the above loop. Among these loop orders, only $L2(i,j,jj,ii)$ exploits coarse- and fine-grain data-level parallelism, respectively, on multiple cores and the vector processing (SIMD) units. In this paper, we will consider each of the loop orders with coarse-grain (SPMD) data-level parallelism that can exploit multiple cores at the appropriate loop level (based on the dependencies in the program) and the fine-grain (SIMD) data-level parallelism, if permissible, at the innermost loop level.

2.1.2 Intel Xeon Phi Knights Landing Architecture

The architecture for the second generation many-cores Intel Xeon Phi product is known as Knights Landing (KNL) (Sodani, 2015; Sodani et al., 2016). It is the first self-boot Xeon Phi processor and is binary compatible with the mainline x86 Instruction Set Architecture. A typical KNL processor comprises a basic unit called a tile consisting of two cores, two vector-processing units (VPUs) per core, and 1MB L2 cache shared between the two cores. Further details about the architecture are available at (Sodani, 2015; Sodani et al., 2016). The KNL processor used in this work has 32 active tiles, 64 cores, and 128 VPUs.

The core of a tile is a two-wide, out-of-order core derived from the Intel Atom processor (Wolfe, 1987) named Silvermont but modified to include many high-performance computing features. The VPU supports all Floating-point computations ranging from legacy instructions to the latest AVX-512 (Xue, 2000) vector instructions. Thus, it supports up to eight double-precision floating-point operations per cycle in each VPU. The tiles are interconnected by a cache-coherent two-dimensional mesh interconnect that provides higher bandwidth and lower latency for communication across tiles. The L2 caches in all tiles are kept coherent using MESIF (Hackenberg et al., 2009) cache-coherent protocol. The mesh supports three modes of clustered operations to provide different levels of address affinity: i) all-to-all mode, ii) quadrant mode, and iii) sub-NUMA clustering mode. Our experiments, use the default quadrant mode as it provides higher bandwidth and lower latency for memory accesses and is transparent to software.

KNL processor has two types of memory: multi-channel DRAM (MCDRAM) and DDR providing high bandwidth and large capacity. There are eight MCDRAM devices integrated on the package, and each is of 2GB capacity, thus providing a total of 16GB of high-bandwidth memory. DDR offers high-capacity memory, which is off package and has two DDR4 memory controllers on opposite sides of the chip, each controlling three channels. The memory can be configured at boot time in one of the three modes: i) Cache mode where MCDRAM is a cache for DDR, ii) Flat mode where MCDRAM is treated like standard memory in the same address space as DDR and iii) Hybrid mode where a portion of MCDRAM is cache and the remaining is flat. In our experiments, we used the cache mode where the MCDRAM serves as the last-level shared cache for all the cores.

KNL supports two types of hardware prefetchers for prefetching data on each tile. The L1 cache prefetcher, also known as Instruction Pointer Prefetcher (IPP), analyzes all accesses in the data cache and instructions and inserts hardware prefetches to the L1 if a strided access pattern is detected on a cacheable page. The L2 hardware prefetcher identifies streaming access patterns and can track up to 48 streams. It prefetches data into the L2 cache whenever a stream is detected. KNL allows to select and set prefetchers at boot time independently. While prefetching, in general, helps improve the performance of an application running on KNL, aggressive prefetching can degrade performance for some applications.

The key opportunities for parallelism on KNL are task- and data-level parallelism. Task-level parallelism could either be SPMD tasks or tasks which perform entirely different computations on different cores. OpenMP (Dagum and Menon, 1998), High Performance Fortran (Koelbel et al., 1994), and Thread Building Blocks (TBB) (Kukanov and Voss, 2007) are the most prevalent standards used to create parallel programs to exploit task-level parallelism on KNL. Vectorization or fine-grain data parallelism can be achieved using Intel libraries, SIMD directives/pragmas to assist or force vectorization, and auto-vectorization by the compiler.

Therefore, it is critical for many-core architectures like KNL to perform appropriate loop transformations that can exploit both coarse- and fine-grain data parallelism utilizing execution on all cores and using all VPUs effectively. Further, the transformed code should also be able to exploit data reuse in L1/L2 caches and the MCDRAM, which is being used as a last-level shared cache. Lastly, the data access patterns in the transformed loop should enable efficient prefetching of data into L1/L2 cache without incurring performance degradation.

2.2 Motivation

This section motivates our work with a few loops taken from the Polybench benchmark suite (Pouchet, 2012; Pouchet and Grauer-Gray, 2011). First, we will consider the tiled version of the `gemver_k2` loop shown in Listing 2 and its performance for different loop orders $L1$-$L6$.

The normalized execution cycles, normalized with respect to the best-performing loop order (one with the lowest execution time), of these loops on Intel Xeon Phi KNL server (The details of our experimental framework and methodology are presented in Section 4), with their hardware prefetchers disabled, are reported in Table 1. For this loop, loop order $L2(i, j, jj, ii)$ is the best-performing order and hence is the *optimal* loop order. Pluto framework (Bondhugula et al., 2008b), which can exploit coarse- and fine-grain data-level parallelism both at the outermost and the innermost levels, picks the $L2(i, j, jj, ii)$ loop order for this loop. This follows the intuition and the heuristic used in the framework. The normalized execution cycles of different loop orders, with all prefetchers enabled, are reported in Table 2. There too we observe a similar performance trend among $L1(i, j, ii, jj)$, $L2(i, j, jj, ii)$, and $L3(i, ii, j, jj)$ although the performance loss incurred by $L1(i, j, ii, jj)$ and $L3(i, ii, j, jj)$ are larger.

Next, consider another loop from the `gemver` benchmark as shown in Listing 3.

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

Listing 3: gemver_k3 loop

```
for  (i=0;i<N-2;i++)
  for  (j=0;j<N;j++)
    X[N-2-i][j]=(X[N-2-i][j]-X[N-i-3][j]
                 *A[N-3-i][j])/B[N-2-i][j];
```

Listing 4: adi_k4 loop

This loop has parallelism in the i-dimension, and the access of 2-dimensional array `A` is row-major ordering.

Loop orders $L1(i,j,ii,jj)$ and $L3(i,ii,j,jj)$ exploit coarse-grain parallelism only at the outermost level, while $L2(i,j,jj,ii)$ exploits data-level parallelism at both the levels but leads to scatter-gather memory accesses (due to `A[ii][jj]` accesses in the innermost loop). Scatter-gather memory accesses enable vector operations on data items that are apart from each other by a constant, non-unit stride. However, they are more expensive than simple vector load/store operations. The normalized execution cycle of loops orders $L1(i,j,ii,jj)$, $L2(i,j,jj,ii)$, and $L3(i,ii,j,jj)$ in Tables 1 and 2 show different trends depending on whether the hardware prefetchers are enabled vs. disabled. When hardware prefetchers are disabled, $L2(i,j,jj,ii)$ performs better than $L1(i,j,ii,jj)$ and $L3(i,ii,j,jj)$ even though it has scatter-gather accesses but exploits data-level parallelism. On the other hand, when prefetchers are enabled, loop order $L3(i,ii,j,jj)$ performs better, indicating a trade-off between data access cost and exploiting data-level parallelism. Last, we note that neither Polly (Grosser et al., 2011) nor Pluto (Bondhugula et al., 2008b) picks the optimal loop order for this loop `gemver_k3`. They both choose loop order $L1(i,j,ii,jj)$, which incurs a performance loss of 12% and 82% depending on whether the prefetchers are enabled or disabled.

Last, we consider a loop nest from `adi` benchmark. The loop in Listing 4 is parallel in j-dimension and accesses three 2-dimensional arrays `A`, `B`, and `X`, and all accesses are of the form `[N-i][j]`. For this loop nest, the loop order $L4(j,i,ii,jj)$, which exploits SPMD and SIMD parallelism at the outermost and the innermost loop levels respectively, is likely to give the best performance. This is, in fact, the case for problem size $N = 4096$ (refer to Table 1). However, for larger values of N ($N = 8192$ or $N = 16384$), loop order $L1(i,j,ii,jj)$ has the lowest execution cycles. Thus, even under the same prefetch configuration (prefetchers disabled), the best performing loop order can change based on the problem size. Note that for this loop with parallelism only at the j-dimension, loop orders L1, L2, and L3, which have the i-dimension at the outermost loop level, incur synchronization cost for each i iteration. Thus, the heuristics used in Pluto (Bondhugula et al., 2008b) or Polly (Grosser et al., 2011) fail, and they end up picking $L4(j,i,ii,jj)$ (or $L5(j,i,jj,ii)$), irrespective of the problem size. This results in a performance of loss of 12% – 215% for $N = 8192$. From the examples presented in Tables 1 and 2, we observe that identifying a suboptimal loop order can significantly increase execution time, by a factor of 1.12x to 38.66x! This reemphasizes the importance of identifying the optimal loop order while applying the polyhedral optimizations.

To summarize, our examples demonstrate that the performance of a loop nest depends on multiple factors, including SPMD and SIMD data-level parallelism exploited, the abilities of the prefetchers, the synchronization overheads incurred, the data access patterns in the loop, and also the size (or volume) of the data accessed. This motivates us to use a simple machine learning technique (specifically SVM classifier) to identify the optimal loop order for a given loop nest.

3 Identifying Optimal Loop Order as a Classification Problem

Before we describe our approach to identifying the optimal loop order, we state the assumptions made in this paper.

3.1 Assumptions

In this paper, we limit ourselves to identifying the optimal loop order for a 2-dimensional perfect loop nest, which is represented in the canonical form as shown in Listing 5:

```
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

where $S1$, $S2$, ... , Sn are simple arithmetic statements involving accesses to one-/two-dimensional arrays and scalars. The array indices are affine expressions involving either of the loop iterators or constants. A pair of statements in the loop can have true, anti-, and output dependencies between them which are either loop-carried or loop-independent. We assume all the dependence distances to be constant. We assume that each loop is tiled with a fixed tile size T. Further, we assume that the tiled loop is permutable and any of the six possible permutations ($L1$, $L2$, $L3$, $L4$, $L5$ and $L6$) result in a legal loop preserving the dependencies in the original loop nest.

Although the above considers only a limited subset of loops, it certainly includes a significant subset of practical / real-world loop nests which are typically targetted by Polyhedral optimizations. Further, while it is possible to relax one or more of the above assumptions, the proposed classification approach, synthetic loop generation, and selection of input features may also need to be extended accordingly. We leave this to future work.

3.2 Synthetic Loop Generation

Our approach of applying supervised learning to find the optimal loop order for a loop nest requires a large number of loops that can be used as training data sets. However, the number of available loops from various benchmark suites such as NAS parallel benchmarks

Table 1 Normalized Execution Cycles with Prefetchers Disabled

	Normalized Execution Cycles						Loop Order Identified by	
Benchmark	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**Polly**	**Pluto**
gemver_k2	2.50	1.00	8.46	3.16	1.62	33.01	L1	L2
gemver_k3	1.82	1.00	1.79	2.43	1.62	25.32	L1	L1
adi_k4(4096)	1.62	3.55	24.39	1.00	2.80	10.91	L5	L4
adi_k4(8192)	1.00	3.21	13.04	1.12	3.15	18.82	L5	L4

Table 2 Normalized Execution Cycles with Prefetchers Enabled

	Normalized Execution Cycles						Loop Order Identified by	
Benchmark	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**Polly**	**Pluto**
gemver_k2	2.39	1.00	9.44	3.49	1.60	38.66	L1	L2
gemver_k3	1.12	1.29	1.00	2.02	2.26	27.94	L1	L1
adi_k4(4096)	1.88	4.66	25.64	1.00	2.74	8.49	L5	L4
adi_k4(8192)	1.75	4.85	18.03	1.00	3.54	13.46	L5	L4

(Barszcz et al., 1991), PARSEC (Bienia et al., 2008), and Polybench (Pouchet, 2012), that are 2-dimensional and tileable are very few perhaps yielding a few 10's of loops. Further, as machine learning techniques are applied to select other parameters (such as the tile size) of loop transformation techniques and architecture configurations like prefetchers, the need to have many loops for training purposes increases. Also, for different loop transformation techniques, the characteristics of the train loops used may be different, resulting in fewer training loops applicable for that purpose. It is desirable to have a method that can generate thousands or even millions of (synthetic) loops which are different (in terms of a number of 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.

In order to address this, we have developed a methodology to generate synthetic loops automatically. 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 a large number of loops. In this work, we have considered *perfect*, 2-dimensional loop nests which can be tiled in either dimension. The synthetic loop generation methodology can be extended to higher dimension loop nests as well.

The generated loops have the following properties:

- Each loop nest is of the form shown in Listing 5 and can have one or more assignment statements involving arithmetic operations. Further, we represent each statement in the canonical 3-address form

  `a = b op c`

  Note that the 3-address form is not restrictive and can represent any complex arithmetic statements, including stencil computations, as a sequence of 3-address statements.

- Each operand in the 3-address statement can be a scalar, 1- or 2-dimensional array with appropriate indices. It is possible to easily extend the method to include higher dimensional arrays.

- 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)]` . For now, we limit our method to affine functions involving a single loop index. This can be extended later, as we extend the dependencies in our loop to non-constant values.

- 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 of type true, anti-, or output dependence. Last, 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 that have dependencies in both dimensions, as they do not permit parallelism in either dimension.

3.2.1 DDG Generator

Each loop can be represented as a directed 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 above 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. This approach helps us to generate a large number of loops just by varying these DDG properties.

The functionality of the DDG generator is described below. 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 type (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 also 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 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 (Pouchet, 2012) suite.

Using the values from the input file, our tool generates a random DDG satisfying the above characteristics. It can generate different 2-dimensional loops which have parallelism in at least one of the dimensions involving 1- or 2-dimensional arrays. For each edge, it assigns a distance-vector randomly using the probability values specified in the input file. In our experiments, we limited the dependence in each dimension from -2 to 2 as they cover a majority of the test loops considered. For an edge, if the assigned distance vector is lexicographically positive, it is considered a true dependence edge. If the distance vector is lexicographically negative, the direction of the edge is reversed, and it represents an anti-dependence edge.

Each node in the DDG represents a statement in the loop and hence is associated with a 1- or 2-dimensional array, which is the destination (left-hand side) of the statement. We do not consider a scalar variable for the destination as it would create dependence in both dimensions of the loop, preventing parallelism. Further, the valid loop transformation (scalar expansion) performed to parallelize the loop, anyway, introduces temporary 1- or 2-dimensional array variables as the destination. The required number of output dependence edges is generated by identifying pairs of nodes associated with the same destination variable. To ensure that the generated DDG corresponds to loops containing statements in canonical 3-address, the random DDG generation limits the in-degree and the out-degree of a node to at most two. Each in-degree of a node corresponds to an input dependence (or source operand) for the node.

We ensure that our tool 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. This is true if every directed cycle in the DDG contains at least one edge with a loop-carried dependence in one or more dimensions. To ensure this, our random DDG generation tool inserts true dependence edge with (0,0) dependence only from lower-numbered nodes to higher-numbered nodes. We omit formal proof of the above claim due to space constraints.

However, the generated DDG may be incomplete in the sense that there could be some implied dependence edges. We note here that such implied edges, as discussed later, do not affect the correctness of the generated C-code. However, our tool identifies the cases where these implied edges need to be added and inserts them in the DDG. We illustrate one such scenario with the following simple example.

Consider a DDG as shown in Figure 1a. This DDG has two nodes and two edges. The dependence distance vector of length 2 indicates that 2-dimensional arrays are involved in dependence relations. The self-edge on S1 is an anti-dependence edge, and there is an output dependence edge from S1 to S2. Due to the output dependence, both S1 and S2 write to the same destination variable. This, in turn, implies that there should be an anti-dependence from S1 to S2, which is what we refer to as an implied dependence, as shown in Figure 1b. We add all such implied dependence edges.

We have implemented our random DDG generator tool in Python. The DDG is represented using the dictionary data structure. An example DDG generated by our tool is shown in Figure 2. Figure 2a shows the DDG generated initially with five nodes and eight edges. True dependence edges are represented using solid black arrows, anti-dependence edges are represented using dashed red arrows, and output dependence edges are represented using dashed blue arrows. Figure 2b shows the DDG generated after the implied edges are added. Output dependence edges between nodes (1,5) and (2,5) imply that there is an output dependency between nodes (1,2). Similarly, an anti-dependence edge between nodes (4,1) implies anti-dependence edges between nodes (4,2) and (4,5) of the same weight. Our tool also generates a visual representation of the graph, as depicted in Figure 2.

3.2.2 Synthesizing C-Code for the Generated DDG

The next step is to generate the loop nest corresponding to the generated DDG with valid C-statements. Since our DDG construction procedure allows loop-independent true data dependencies only from lower-numbered to higher-numbered nodes, it can easily be shown that the node numbering dictates a valid ordering of the

Figure 1: Data Dependency Graphs

(a) (b)

statements in the loop. Thus, for each node in the DDG, we generate the corresponding C-statement in the order of the node number, with the destination variable same as that associated with the node. The source operands are variables associated with the source nodes of each input edge and the distance vector, determining the index of the source vector. For nodes having an in-degree less than 2, the required additional inputs are assumed to be variables (scalar, 1- or 2-dimensional array), not modified (read-only) in the loop. The statements of the loops are enclosed with a 2-dimensional nested for loop.

For the DDG shown in Figure 2a, our synthesizer generates the C-code shown in Listing 6. Our tool can generate programs involving any number of nodes and edges. In our training data set, we have constructed programs with up to 35 statements and more than 70 dependence edges. These sizes correspond to loops encountered in real-world programs.

```
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

3.2.3 Validating the Synthesizer

In order to ensure that our tool generates valid C-statements for a given DDG, we implemented a small validation tool using the ISL (Integer Set Library) (Verdoolaege, 2010) tool. The ISL tool takes the set of C-statements generated by the synthesizer as inputs and outputs all the dependencies between those statements. The DDG constructed from all the dependence relations given as output by the ISL tool is then compared with the random DDG generated by our tool (along with the implied dependencies). For the synthesized listing shown in Listing 6, the DDG generated by the ISL tool is shown in Figure 2c, which matches well with Figure 2b.

3.3 Identifying the Optimal Loop Order for a Loop Nest

As mentioned earlier, when a 2-dimensional loop is tiled in each dimension, it results in six possible permutations (henceforth referred to as loop orders). We consider only loop nests in which all six loop orders are legal. Depending on the number of dependencies present in the loop nest, either the i-loop or the j-loop or both could be parallel. While it is possible to have a loop in which parallelism is not present in either dimension, such loops are interesting for a loop-transformation study on many-core architectures and hence are not considered in our discussion. Depending on the parallelism available in the loop nest, a specific loop order will exploit the coarse-grain data-level parallelism (SPMD) in the respective inter-tile dimensions of the tiled loop. Fine-grain data-level parallelism (SIMD) is exploited if the corresponding intra-tile dimension happens to be the innermost dimension. For example, if the original loop nest is parallel (only) in i-dimension, then in loop order $L1(i, j, ii, jj)$, SPMD parallelism is exploited in the outermost loop. However, SIMD parallelism cannot be exploited as ii dimension is not at the innermost level. As another example, for the same loop, loop order $L6(j, jj, i, ii)$ exploits SPMD parallelism at the i-dimension (second innermost loop), with appropriate synchronization, included for each jj-iteration. Further, for $L6(j, jj, i, ii)$ loop order, SIMD parallelism is exploited at the innermost loop level. We use Polyhedral techniques and the Polly framework (Grosser et al., 2011) to identify and exploit the parallelism for the different loop orders.

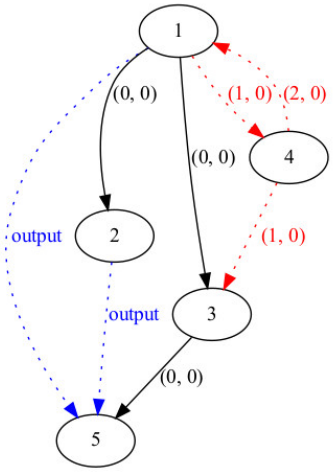
Identifying the optimal loop order for a given loop nest can be formulated as a supervised learning problem. The loop nests are characterized by a few features, referred to as the input features of the loop. We propose using the following as input features:

1. The dimension(s) of the original loop nest, i.e., i or j or both i and j, that has (have) parallelism.

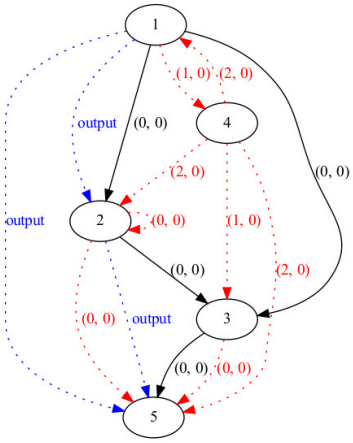
2. The number of references to 2-dimensional arrays of the form `A[f(i)][g(j)]`, where `f` and `g` are affine functions.

3. The number of references to 2-dimensional arrays of the form `A[f(j)][g(i)]`, where `f` and `g` are affine functions.

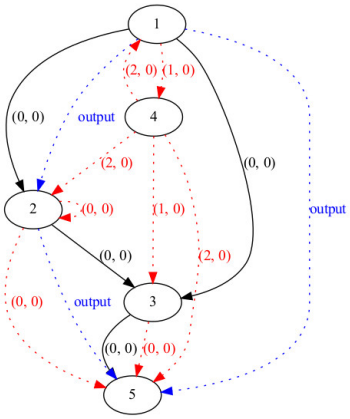
Figure 2: Different DDGs generated



(a) Initial DDG



(b) DDG after adding implied edges



(c) DDG using ISL tool dependence relations

4. The number of references to 1-dimensional arrays of the form `A[f(i)]`, where `f` is an affine function.
5. The number of references to 1-dimensional arrays of the form `A[f(j)]`, where `f` is an affine function.
6. The number of 2-dimensional arrays of the form `A[f(i)][g(j)]`, where `f` and `g` are affine functions.
7. The number of 2-dimensional arrays of the form `A[f(j)][g(i)]`, where `f` and `g` are affine functions.
8. The number of 1-dimensional arrays of the form `A[f(i)]`, where `f` is an affine function.
9. The number of 1-dimensional arrays of the form `A[f(j)]`, where `f` is an affine function.
10. Value of N

These features are carefully selected, using domain knowledge, from a host of features that characterize the performance of the given loop nest in various loop orders. Other features, such as the number of integer/floating-point operations in the loop nest and the number of synchronization operations required, were also considered for inclusion in the input feature. However, we discarded them, either after some initial experimentation or after careful consideration, to keep the input feature vector as small as possible. For example, the number of integer operations, though not directly represented as a feature in our approach, is indirectly represented by the number of 1- and 2-dimensional array references. We note that the selected input features for a loop nest can be easily extracted by a compiler. The output label for the loop nest is the best performing loop order, i.e., one of $L1$ to $L6$. The problem on hand is:

> Given the input features of a (test) loop nest, the problem is to identify the output class L, which represents the best performing loop order for the loop nest.

This is naturally formulated as a classification problem. The classifier is built using a set of training data (loop nests) whose output classes are already labelled. The features of the training data loops form the input feature vector. We propose to use the SVM classifier (Meyer and Wien, 2015; Meyer et al., 2019) for this. SVM is a supervised machine learning algorithm that can be used for classification problems. 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 (Meyer and Wien, 2015; Meyer et al., 2019). In our work, we propose to use a multi-class SVM classifier that uses the linear kernels. The classifier is trained on the train data (loop nests from our loop generator tool). The developed classifier is then used on test loops, for which the input features are available to predict the optimal loop order.

4 Experimental Methodology

4.1 Architecture Used

We have used an Intel Xeon Phi Knights Landing (KNL) system (Sodani, 2015; Sodani et al., 2016) for our experiments. It has 32 tiles with 64 cores and 128 VPUs. The 64 cores are present on a single socket. The number of threads per core is 4. It has L1d and L1i cache each of capacity 32KB and 16-way 1024KB L2 cache. The 16GB direct-mapped MCDRAM is in cache mode. The architecture details have been discussed in detail in Section 2.2.

4.2 Experimental Methodology

The optimal loop order of a given loop nest is the one that has the lowest execution cycles. To get the execution cycles for loop orders $L1$-$L6$ of each loop, we perform direct measurements on the KNL system using *_rdtsc* function (Yason, 2007). However, when repeated experiments were conducted to measure the execution cycles of a loop, we observed high variation in execution cycles across different instances (in a single execution) and the different executions. The coefficient of variation (CV) across the averages of different executions was in the range of 6%-10%, an unreasonably high figure. These variations in execution cycles are due to Operating System (OS) jitter caused by OS process scheduling policies, handling of interrupts, thread migration (Mazouz et al., 2011; Pusukuri et al., 2012), other processes/daemons running on the system, sharing of resources across processes, etc. In order to minimize these variations, we have incorporated the following restrictions in our KNL system.

- *Single user mode operation*: During the experiments, the KNL machine was entirely dedicated to a single user to ensure no other user processes ran concurrently on the system. This would otherwise influence the measurements due to sharing of resources like last-level caches.

- *Disable hyperthreading*: On each KNL core, hyperthreading was disabled to minimize OS-related latencies due to activities such as thread migration (Pusukuri et al., 2012) or context switches.

- *Disable huge pages*: Support for transparent huge pages (Arcangeli, 2010) was disabled to minimize the latency spikes due to the allocation and maintenance of huge pages.

- *Disable DVFS*: Dynamic voltage frequency scaling was turned off by setting the scaling governor to `performance` and frequency of each core to 1300MHz to minimize dynamic frequency-related variations (Acun et al., 2016).

- *Disable other services*: Services like *cupd* used for printing are disabled to minimize their influence on execution cycles.

Further, we use the following procedure to reduce the variations in the execution cycles across multiple executions of a train loop (synthetic loops generated by our method) or the test loop (Polybench loops). In a single execution, we run the program 20 times and measure the execution cycles in each execution for the entire loop. From these values of a single execution, outliers are removed using the standard $1.5 \times InterQuartileRange(IQR)$ (Rousseeuw and Hubert, 2011) method. The resulting values are accepted if the coefficient of variation (CV) is less than or equal to 2%. If we find that CV is greater than 2%, 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 whole run is discarded if the CV is found to be greater than 2% or if we find more than 20% of the outliers removed anytime. We then 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.

4.3 Benchmarks Used

We have used 2-dimensional perfect loop nests from Polybench-3.2 (Pouchet, 2012) benchmark suite that are permutable, tileable in both dimensions, and have parallelism in at least one dimension. There are 21 such loops. These loops have one or more statements (up to five statements) but contain complex arithmetic expressions involving multiple operations and operands. The name of the benchmark and the number of loop nests taken from them are listed in Table 3. We refer to this set of loops as Poly_Orig loops. In order 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 original form and its 3-address form performs the same operations and produces the same output; only the representation has changed. The introduction of temporary arrays may influence the optimal loop order due to increased memory footprint/working set and the associated cache behaviour. The canonical form also changes the features of the loop nest, which in turn helps us provide additional test cases for our methodology. We refer to this set of loops as Poly_3-AddressCode or Poly_3AC for short.

We have used three different N (problem size) values, namely $N = 4096$, $N = 8192$, and $N = 16384$ in our evaluation. We identify the optimal loop order for each of these loop nests by running these six different loop orders on the Intel Xeon Phi KNL system and selecting

Table 3 List of Benchmarks Used

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

the one with the lowest execution cycles. The optimal loop for a given loop nest specifies its output class.

4.4 Support Vector Machine Classifier

As explained in Section 3, we have used Support Vector Machines (SVM)(Cortes and Vapnik, 1995) as a classifier for identifying the optimal loop order. Classifying a given data point into one of the classes is a multi-class problem with *six* output classes. The SVM implementation (e1071) (Meyer et al., 2019; Meyer and Wien, 2015) solves a multi-class classification problem using 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.

For all our experiments and results discussed in the next section, the train and test data together were normalized to zero mean and unit variance. We have explored both linear and radial basis function kernels of the e1071 package of R(version 3.4.0) for our experiments. However, SVMs with linear kernels were found to perform better on our data set. Regularization hyperparameter in the objective function of the SVM classification problem was tuned using the cross-validation technique. This hyperparameter was varied over the range 2^{-5} to 2^5. The hyperparameter value, which resulted in the best validation set performance, was used for final training and prediction purposes. We have used 5-fold cross-validation to avoid selection bias of the model. Further, for all our experiments, we have fixed the seed value in order to get repeatable results, and we have conducted all classification experiments for two different seed values. However, we are reporting the results for one seed value due to space constraints. The performance numbers for another seed are either same or differ marginally by a small value.

5 Results and Discussions

In this section, we discuss the results of our experiments. We have identified the optimal loop order for the KNL architecture with two hardware prefetch configurations: one where all of the prefetchers are enabled (All_Enable) and another where none of the prefetchers are enabled (None_Enable). In our experiments, we compare the performance of the proposed SVM-based optimal loop order predictor, henceforth referred to as *SVM_Pred*, with other compiler approaches such as Polly (Grosser et al., 2011) and Pluto (Bondhugula et al., 2008b). The main performance metric that we report for all the methods is the performance loss (from optimal), which is an indication of how far away is the performance of the identified loop order from the optimal one.

$$Perf.Loss_{SVM\_Pred} = \left(\frac{Exec.Cycles_{SVM\_Pred}}{Exec.Cycles_{Opt}} - 1 \right) * 100$$

where $Exec.Cycles_{SVM\_Pred}$ and $Exec.Cycles_{Opt}$ refer to the execution cycles of the loop order chosen by SVM_Pred and the optimal loop order of the loop nest, respectively. Further, when reporting the performance for a specific set of test loops, we take the arithmetic mean of the performance losses across different loops in the test set. In addition to performance loss, we also report prediction accuracy of the method, which is defined as the percentage of cases in which the optimal loop order is correctly identified by our classifier or the compared optimizers.

5.1 Predicting Performance of Real Benchmarks

First, we demonstrate the efficacy of SVM_Pred on predicting the optimal loop order with training data consisting of synthetic loops generated using our synthetic loop generator tool (discussed in Section 3) and test cases having Polybench loops either in the original form (Poly_Orig) or the 3-address form (Poly_3AC). We have generated 1503 synthetic benchmarks, which are 2-dimensional loops, perfectly nested, and have parallelism in either or both dimensions and used them as training data set. The characteristics of these benchmarks are based on real-world programs and are specified as the input configuration file for the benchmark generator program. Table 4 shows the performance of SVM_Pred compared with Polly and Pluto. We present the performance numbers for both All_Enable and None_Enable prefetch configurations. Columns 3 – 5 in Table 4 represent the performance loss (relative to optimal loop order), and columns 6 – 8 represent the prediction accuracy.

We observe that for All_Enable prefetch configuration (default setting in the BIOS/system), SVM_Pred performs well, resulting in a performance loss of 4.23% and 1.30% and prediction accuracy of 88.89% and 93.65% for Poly_Orig and Poly_3AC, respectively. In comparison, Pluto incurs a performance loss of 54.25% and 64.59% for Poly_Orig and Poly_3AC loops, respectively, while Polly's performance loss is even higher (98.01% – 127.13%). Polly does not take data-level parallelism due to vectorization into account wherever

Table 4 Performance using Synthetic Benchmarks as Training Data

		Average Performance Loss			Prediction Accuracy		
Test Set	**Prefetch Configuration**	**SVM_Pred (%)**	**Polly (%)**	**Pluto (%)**	**SVM_Pred (%)**	**Polly (%)**	**Pluto (%)**
Poly_Orig	All_Enable	4.23	98.01	54.25	88.89	4.76	19.05
Poly_3AC	All_Enable	1.30	127.13	64.59	93.65	9.52	17.46
Poly_Orig	None_Enable	8.74	53.53	18.45	68.25	9.52	22.22
Poly_3AC	None_Enable	1.82	56.21	11.48	84.13	14.29	15.87

possible. Therefore, it fails to identify the optimal loop order for most of the loop nests of Poly_Orig and Poly_3AC and hence incurs lower prediction accuracies and higher performance losses.

Even for the None_Enable prefetch configuration, SVM_Pred performs well and incurs a performance loss of 8.74% and 1.82% and prediction accuracy of 68.25% and 84.13% for the Poly_Orig and Poly_3AC loops, respectively. In comparison, Pluto incurs a performance loss of 18.45% and 11.48% for Poly_Orig and Poly_3AC loops, while Polly's performance loss is 22.22% and 15.87%.

Though the prediction accuracy of SVM_Pred for None_Enable configuration is not as high as that for All_Enable, the performance loss is about 9%. Our analysis of results for None_Enable configuration indicated that, for some of the test loops where our classifier SVM_Pred mispredicts the optimal loop order, it predicts the second-best performing loop order and thus incurs lower performance loss values. The time required to train SVM_Pred using the entire traindata set is less than a minute on an Intel Core i7 processor running at 2.90 GHz.

5.2 Performance of SVM_Pred using Smaller Train Set

The performance results reported in Table 4 use the entire 1503 loops as train data set. In this subsection, we study whether similar prediction accuracy and performance loss can be achieved consistently by a classifier built with a smaller train data size.

We conducted these experiments with training data set sizes starting from 600 to 1400 in steps of 100. For each train set size, we repeated the experiment with different train data sets chosen randomly from the entire train data. We also experimented with choosing the loops in the train data such that the distribution of $L1 - L6$ output class in the chosen subset match that of the entire train data. We refer to the former train set selection method as random-unconstrained (*rand-uc* for short) and the latter as random with matching class distribution (*rand-mcd* for short). Note that neither of these training data selection procedures requires knowledge of the output classes (optimal loop orders) of the test data loops.

Table 5 presents the results of our study when using train data sets of sizes 600 to 1400. For each data set size, we repeat the experiments ten times with different subsets of data selected using either the *rand-uc* or *rand-mcd* approach. We report the average, minimum and maximum values for performance loss and prediction accuracy (expressed in percentage) across the ten runs. As before, we use both the Poly_Orig and the Poly_3AC test sets in these experiments.

Our experiments show that with smaller train data set, the performance losses with *rand-uc* and *rand-mcd* vary from 4.23% – 58.51% for Poly_Orig and from 1.30% – 30.34% for Poly_3AC, while the prediction accuracy ranges 77.78% – 88.89% and 84.13% – 93.65% for Poly_Orig and Poly_3AC, respectively. Interestingly, models built from different train data sets (of the same size) perform differently, some achieving good prediction accuracy while a few incur larger prediction errors. Thus while the *rand-uc* and *rand-mcd* approaches perform well with smaller data sets, they are not consistent across different train sets of the same size. The results for the None_Enable configuration show similar behavior. Our analysis reveals that the variations in the performance are often caused by the inclusion/exclusion of some synthetic loops in the train data. In order to build more robust models with improved average prediction performance and to reduce the variance of the prediction errors, even with smaller train data set, we propose to use an ensemble approach by building multiple classifiers with different train data sets (of the same size) and using a simple majority voting mechanism to decide the final output class. We will call this ensemble classifier *SVM_Ensemb*.

Next, we conduct experiments to identify the minimum train data size and the number of models required for the ensemble approach.

5.2.1 Determining the Size of Train Data Set with Ensembling

In order to determine the minimum train data set size required with ensembling, we keep the number of models to be ensembled as ten and build classifiers with train data size from 600 to 1400, in steps of 100. Once again, we experimented with selecting the train set with *rand-uc* and *rand-mcd* approaches.

For each train data size, we build ten ensemble predictors, each consisting of ten classifiers; i.e., for each train data size, we build ten different *SVM_Ensemb* predictors and evaluate the performance loss and prediction accuracy on the test sets, namely Poly_Orig

Table 5 Performance of *SVM_Pred* for Different Sized Train Data Sets

Train Data Selection Approach	Metric/ Test Set/ Config.		Train Data Size								
			600	**700**	**800**	**900**	**1000**	**1100**	**1200**	**1300**	**1400**
rand-uc	**Perf. Loss/ Poly_Orig/ All_Enable**	**Avg.**	13.03	10.36	16.44	12.45	7.94	13.08	7.11	5.73	5.97
		Min.	4.23	4.23	4.23	4.23	4.23	4.23	4.23	4.23	4.23
		Max.	31.96	26.57	58.51	30.62	10.69	33.33	10.69	10.31	10.69
	Pred. Acc./ Poly_Orig/ All_Enable	**Avg.**	86.82	86.82	85.87	86.67	87.30	86.51	87.46	88.25	88.10
		Min.	82.54	85.71	80.95	84.13	84.13	79.37	85.71	85.71	85.71
		Max.	88.89	88.89	88.89	88.89	88.89	88.89	88.89	88.89	88.89
	Perf. Loss/ Poly_3AC/ All_Enable	**Avg.**	3.96	1.30	4.00	6.59	1.36	6.57	1.30	1.31	1.31
		Min.	1.30	1.30	1.30	1.30	1.30	1.30	1.30	1.30	1.30
		Max.	27.80	1.34	27.68	27.68	1.92	27.68	1.30	1.33	1.33
	Pred. Acc./ Poly_3AC/ All_Enable	**Avg.**	93.65	93.02	93.17	92.54	93.33	92.54	93.17	93.33	93.65
		Min.	93.65	88.89	88.89	88.89	92.06	88.89	88.89	92.06	93.65
		Max.	93.65	93.65	93.65	93.65	93.65	93.65	93.65	93.65	93.65
rand-mcd	**Perf. Loss/ Poly_Orig/ All_Enable**	**Avg.**	14.45	17.95	9.54	13.57	10.46	8.75	6.86	7.01	5.62
		Min.	4.23	4.23	4.23	4.23	4.23	4.23	4.23	4.23	4.23
		Max.	37.48	32.41	30.62	32.35	37.48	31.57	10.69	13.87	10.69
	Pred. Acc./ Poly_Orig/ All_Enable	**Avg.**	85.24	84.92	87.78	85.24	87.46	87.78	88.10	87.78	88.10
		Min.	77.78	79.37	84.13	79.37	80.95	82.54	85.71	85.71	85.71
		Max.	88.89	88.89	88.89	88.89	88.89	88.89	88.89	88.89	88.89
	Perf. Loss/ Poly_3AC/ All_Enable	**Avg.**	6.85	11.86	3.96	6.85	3.95	3.94	1.30	1.30	1.30
		Min.	1.30	1.30	1.30	1.30	1.30	1.30	1.30	1.30	1.30
		Max.	29.05	27.68	27.80	30.34	27.72	27.68	1.30	1.33	1.33
	Pred. Acc./ Poly_3AC/ All_Enable	**Avg.**	92.38	91.43	92.06	91.90	92.38	92.86	93.65	93.33	93.49
		Min.	88.89	88.89	84.13	84.13	87.30	88.89	93.65	92.06	92.06
		Max.	93.65	93.65	93.65	93.65	93.65	93.65	93.65	93.65	93.65

and Poly_3AC for both All_Enable and None_Enable configurations.

We show the results as scatter plots in Figure 3 for *rand-uc* All_Enable configuration for Poly_Orig test set and *rand-mcd* All_Enable configuration for Poly_3AC test set. The plots for other combinations are similar and are omitted due to space constraints. These plots show the performance of the model for ten runs along with the average of ten runs (as filled blue triangle) for each train data size. As can be seen from the plots, *SVM_Ensemb* prediction results in low-performance loss and high prediction accuracy consistently across different train data sizes greater than 900.

5.2.2 Determining the Number of Models to be Ensembled

Next, we determine the minimum number of models required for *SVM_Ensemb*. For this experiment, we keep the train data set size as 900 and increase the number of ensembles from 2 to 20. Each experiment is repeated ten times, with ten different sets of train data points selected from the synthetic loops using either *rand-uc* or *rand-mcd* approach for the train data selection. The test sets are Poly_Orig and Poly_3AC. Both All-Enable and None-Enable configurations are tried.

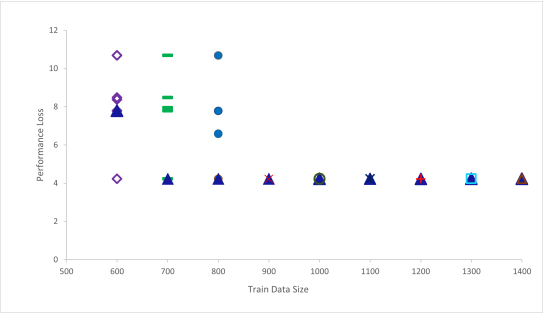
We present the performance loss results for *rand-uc* All_Enable configuration and *rand-mcd* None_Enable configuration for Poly_Orig test set in Figure 4. The performance of *SVM_Ensemb* reaches good prediction accuracy and low-performance loss for a number of models greater than 8. The achieved performance loss and prediction accuracy numbers are comparable to those reported for *SVM_Pred* in Table 4.

5.3 Generalization Ability of the Proposed Classifier

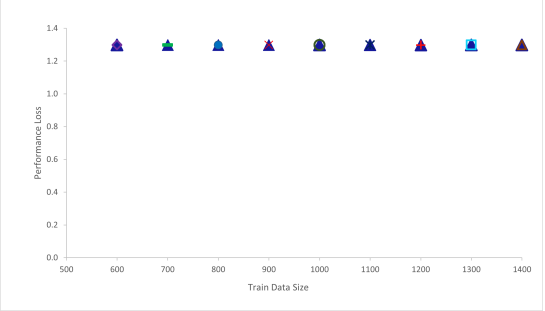
Until this point, all of the classification results in this paper were computed using a test set of actual (i.e., not synthetic) loops from Poly_Orig or Poly_3AC sets. In this section, we derive an out-of-bag (OOB) score for *SVM_Ensemb*. The OOB score gives an unbiased estimate of the generalization ability of the classifier as it does not make use of any test data set and can be used to validate an ensemble model. For a classifier with train data set T_r, where $T_r \subset T$, the set $T - T_r$, i.e., the set of all training data points that are not part of this classifier's training set forms the out-of-bag data. For every data point present in the training data set, we find the class label of this data point using all the trained classifiers in the ensemble whose training set did not contain this data point. Then the final prediction is obtained based on voting.

To find the OOB score of our classifier, we built an ensemble of ten classifiers. The experiments were conducted for training data set size of 600, 700, and 800, and the experiments were repeated five times across

Figure 3: Performance of *SVM_Ensemb* for Different Sized Train Data Sets

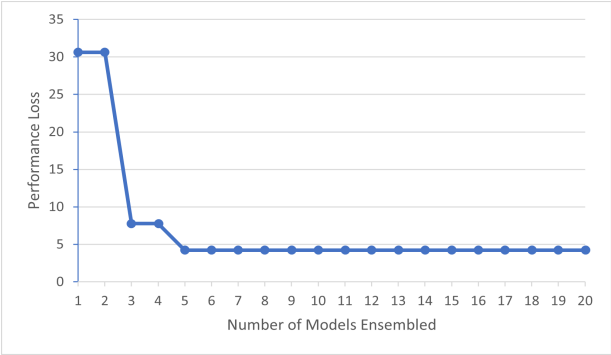


(a) *rand-uc* Poly_Orig All_Enable Configuration

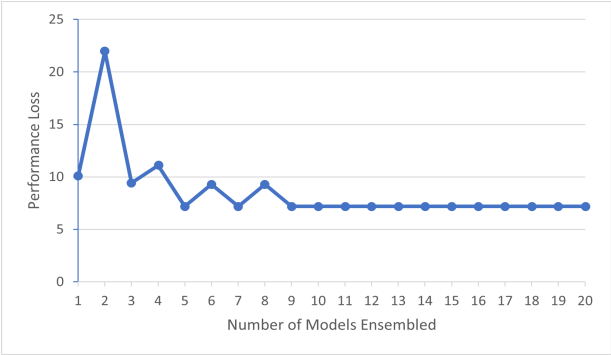


(b) *rand-mcd* Poly_3AC All_Enable Configuration

Figure 4: Performance of *SVM_Ensemb* for Different Number of Models Ensembled



(a) *rand-uc* Poly_Orig All_Enable Configuration



(b) *rand-mcd* Poly_Orig None_Enable Configuration

each size with different train data set chosen randomly to ensure that performance does not depend on a specific train data set.

Table 6 presents the performance loss values and OOB score, which is the prediction accuracy of our classifier for OOB data points for All_Enable and None_Enable configurations for three different training data set sizes. From the table, it can be observed that OOB scores of over 88% for All_Enable and 83% for None_Enable configurations were achieved by the ensemble method. The average performance loss is only 4.32% for All_Enable and 4.38% for None_Enable configuration, indicating a good generalization ability of our classifier for any test data set. It was observed that, on an average, 7 out of the 10 classifiers participated in the voting for each out-of-bag data point.

6 Related Work

In this section, we will review the literature related to Polyhedral loop transformations and those that apply Machine Learning techniques in compilers.

Polyhedral Loop Transformations: Most of the work on loop transformations is towards coming up with a compiler framework based on the polyhedral model, used to explore the sequence of loop transformations for the underlying architecture and provide opportunities for other transformations.

Girbal et al. (Girbal et al., 2006) present a framework that uses a unified representation of loops and statements to support program transformations such as loop fusion, tiling, array forward substitution, statement reordering, array padding, etc., and compositions of these transformations. The proposed techniques have been implemented in the Open64/ORC/EKOPath family of compilers. However, our approach looks at parallelization and vectorization opportunities for a given loop nest. Trifunovic et al. (Trifunovic et al., 2009) present a fast and accurate cost model and a framework to extract vectorization opportunities using polyhedral representation and select an optimal strategy among them.

Pouchet et al. (Pouchet et al., 2011) propose the decomposition of the optimization problem represented as convex polyhedron into sub-problems of much lower complexity, introducing *fusibility* concept. They propose a scheduling algorithm to explore the pruned convex polyhedron. Their automatic optimization and parallelization approach has been implemented in PoCC (Pouchet et al., 2019). Bondhugula et al. (Bondhugula et al., 2008a) develop a framework for automatic parallelization and data locality optimization of imperfectly nested loops in the polyhedral model. The framework finds tiling hyperplanes that minimize inter-tile communication volume in coarse-grained parallelism and facilitate data locality by minimizing reuse distances of C/Fortran code. Ansel et al. (Ansel et al., 2014) propose an open-source framework called OpenTuner for building domain-specific multi-objective program autotuners. It ensembles search techniques to program autotuning to find an optimal solution. The framework is used to implement autotuners for tuning GCC/G++ flags, Halide DSL, etc.

A framework for integrated data locality, multi-core parallelism, and SIMD execution of programs was proposed in (Kong et al., 2013). Data locality is achieved by generating tiled code with a data footprint smaller than L1 cache and is decomposed into *codelets*, targetting parallelization, data reuse, alignment, SIMD execution, and stride of memory references. Sioutas et al.(Sioutas et al., 2018) 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. It selects the loop order that minimizes the distance between the corresponding inter and intra-tile loops.

The design and implementation of an automatic polyhedral source-to-source transformation framework that can optimize regular programs for both parallelism and locality has been proposed in (Bondhugula et al., 2008b). It is 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. Grosser et al.(Grosser et al., 2011) implement polyhedral techniques on top of the LLVM framework to transform parts of the program in a language-independent way. This infrastructure supports a wide range of programming languages and automatically optimizes them. While (Grosser et al., 2011) 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 (Bondhugula et al., 2008b) and (Grosser et al., 2011).

Machine learning approaches: Next, we review literature that use machine learning (ML) based approaches to predict the sequence of transformations or rank different variants of a program to improve the performance on a given architecture.

Stock et al. (Stock et al., 2012) develop an ML model trained using 30 randomly generated loops called tensor contractions (TCs). The optimizations like loop permutation, vectorized loop, and unroll and jam are considered. The offline trained model predicts the performance of several transformed variants of the input test program to select the best variant. We adopt a different approach as we consider the hardware prefetch configuration of the system and propose a tool to generate synthetic benchmarks by varying the properties of a DDG. In (Agakov et al., 2006), the authors develop an ML model by training a set of programs to identify the shape of the search space for iterative optimization. They consider transformations like loop unrolling, common subexpression elimination, if hoisting, and copy propagation. However, our approach considers optimization space of parallelization, vectorization and data locality transformations.

Table 6 Performance of the Classifier for OOB Data Points

Runs	All_Enable						None_Enable					
	Train Data Size						Train Data Size					
	600		700		800		600		700		800	
	Perf. Loss %	OOB Score %	Perf. Loss %	OOB Score %	Perf. Loss %	OOB Score %	Perf. Loss %	OOB Score %	Perf. Loss %	OOB Score %	Perf. Loss %	OOB Score %
Run1	4.41	88.62	4.23	88.96	4.53	88.96	4.30	83.30	4.43	83.30	4.25	83.63
Run2	4.31	88.82	4.33	88.76	4.12	89.16	4.65	82.83	4.32	82.90	4.21	83.17
Run3	4.16	88.82	4.22	88.89	4.75	88.82	4.57	83.03	4.21	83.50	4.14	83.30
Run4	4.06	88.69	4.31	88.89	4.37	88.69	4.24	83.03	4.55	83.03	4.79	82.10
Run5	4.40	88.76	4.22	88.82	4.34	88.96	4.39	82.83	4.28	83.63	4.35	83.57

Dubach et al. (Dubach et al., 2009) apply ML techniques to develop an optimizing compiler for embedded systems that maps a microarchitecture description plus the hardware performance counters from a single run of the program to the best compiler optimization passes. Cosensa et al. (Cosenza et al., 2017) model the performance of stencil computations with an SVM model built using train data of 60 different stencil codes generated using the code generator. Each instance of the stencil is executed with a varying number of randomly generated tuning vectors (tile size, loop unrolling factor and the chunk size), generating a rank for each of them and are used as input by the ordinal regression algorithm that ranks these tuning configurations.

An autotuning compiler framework has been proposed in (Ashouri et al., 2016), using Bayesian Networks for embedded ARM-based platforms. A statistical model based on Bayesian Network correlates application's features to compiler optimizations consisting of loop unrolling, function inlining, induction variable optimization on trees, among others. Ashouri et al. (Ashouri et al., 2017) apply ML techniques to predict the phase order of compiler optimization sequences. Different compiler optimizations together with the reduced program features, and are fed to an ML algorithm to model the speedup predictor that selects the best set of compiler optimization sequences. Synthesizing C programs using a combination of web crawling and type inference is proposed in (da Silva et al., 2021). The framework gathers C source-codes from the GitHub repository, and the type inference engine makes them compilable by filling the missing pieces.

Cummins et al. (Cummins et al., 2017) propose CLgen to mine open-source repositories for OpenCL program fragments and apply deep learning techniques to construct models for generating thousands of human like programs automatically. Haj-Ali et al. (Haj-Ali et al., 2020) apply deep reinforcement learning (RL) for handling loop vectorization. They explore random search, supervised learning methods, and supervised fully connected neural networks based on RL to come up with an auto-vectorization method. Stephenson et al. (Stephenson and Amarasinghe, 2005) apply supervised classification to predict the unroll factors for loops taken from various suites. The methodologies reported in the literature do not generate nested loops and loops which have parallelism in either one or both dimensions and hence cannot be adopted for the current work.

We also use a machine learning approach to predict an optimal loop order permutation for a given loop in a test data set. We observe that the performance of a loop also depends on the hardware prefetch configuration of a system. The optimal loop order for a given loop order may be different under different prefetch configurations, and our model can capture it. We also propose a tool to generate synthetic loops of specifiable characteristics to be used as train data set. Our tool can generate a large corpus of loops which will solve the problem of inadequate training data sets often faced in applying machine learning techniques for compiler optimizations.

7 Conclusion

In this work, we proposed a technique for identifying the optimal loop order for a given loop nest using a supervised machine learning approach for Intel KNL architecture. Our approach builds a Support Vector Machine (SVM) based ensemble model of classifiers that maps input features to target output class based on train data set. We also developed a tool to generate synthetic loops, modelled with parameterized characteristics, representing real-world loops. Our proposed tool can generate synthetic loops with required characteristics that can be used as train data set in building classifiers. Our proposed ensemble model identifies near-optimal loop orders which are within 4.23% and 8.74% of the optimal loop order for the two prefetch configurations studied and outperforms state-of-the-art techniques, Pluto and Polly by 50.02%, 9.71% and 93.78%, 44.79%, respectively. The current approach can be extended to include more input features to build a classification/predictive model for predicting the optimal prefetch configuration or tile size for a given loop nest.

References

B. Acun, P. Miller, and L. V. Kale. Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016.

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, J. Thomson, M. Toussaint, and C. K. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11–pp. IEEE, 2006.

J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 233–246, 1984.

H. Amiri and A. Shahbahrami. Simd programming using intel vector extensions. *Journal of Parallel and Distributed Computing*, 135:83–100, 2020.

J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

A. Arcangeli. Transparent hugepage support. In *KVM forum*, volume 9, 2010.

A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, and C. Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–25, 2016.

A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos. Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–28, 2017.

E. Barszcz, J. Barton, L. Dagum, P. Frederickson, T. Lasinski, R. Schreiber, V. Venkatakrishnan, S. Weeratunga, D. Bailey, D. Browning, et al. The nas parallel benchmarks. In *The International Journal of Supercomputer Applications*. Citeseer, 1991.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008a.

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, 2008b.

T. W. Burger. Intel multi-core processors: Quick reference guide. *Intel Corporation*, 2005.

M. Cornea. Intel avx-512 instructions and their use in the implementation of math functions. *Intel Corporation*, pages 1–20, 2015.

C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

B. Cosenza, J. J. Durillo, S. Ermon, and B. Juurlink. Autotuning stencil computations with structural ordinal regression learning. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 287–296. IEEE, 2017.

C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99. IEEE, 2017.

A. F. da Silva, B. C. Kind, J. W. de Souza Magalhães, J. N. Rocha, B. C. F. Guimaraes, and F. M. Q. Pereira. 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)*, pages 378–390. IEEE, 2021.

L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 78–88, 2009.

P. Gepner and M. F. Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, pages 9–13. IEEE, 2006.

S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International*

Journal of Parallel Programming, 34(3):261–317, 2006.

T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture*, pages 413–422, 2009.

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

J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. *white paper, Intel*, 2006.

C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr, and M. Zosel. *The high performance Fortran handbook*. MIT press, 1994.

M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 127–138, 2013.

A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

C. Lomont. Introduction to intel advanced vector extensions. *Intel white paper*, 23, 2011.

A. Mazouz, S. Touati, and D. Barthou. Analysing the variability of openmp programs performances on multicore architectures. In *Fourth workshop on programmability issues for heterogeneous multicores (MULTIPROG-2011)*, page 14, 2011.

D. Meyer and F. Wien. Support vector machines. *The Interface to libsvm in package e1071*, 28:20, 2015.

D. Meyer, E. Dimitriadou, K. Hornik, A. Weingessel, F. Leisch, C.-C. Chang, C.-C. Lin, and M. D. Meyer. Package 'e1071'. *The R Journal*, 2019.

L. Pouchet. Polybench: The polyhedral benchmark suite. `http://www. cs. ucla. edu/pouchet/software/polybench`, 2012.

L. Pouchet and S. Grauer-Gray. Polybench: The polyhedral benchmark suite (2011), version 3.2. `http://www-roc. inria. fr/ pouchet/software/polybench`, 2011.

L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices*, 46(1): 549–562, 2011.

L.-N. Pouchet, C. Bastoul, and U. Bondhugula. Pocc: the polyhedral compiler collection. `http://web.cs.ucla.edu/ pouchet/software/pocc/`, 2019.

K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread tranquilizer: Dynamically reducing performance variation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

P. J. Rousseeuw and M. Hubert. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery*, 1(1):73–79, 2011.

S. Sioutas, S. Stuijk, H. Corporaal, T. Basten, and L. Somers. Loop transformations leveraging hardware prefetching. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 254–264, 2018.

A. Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24. IEEE, 2015.

A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *International symposium on code generation and optimization*, pages 123–134. IEEE, 2005.

K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–23, 2012.

K. Trifunovic, D. Nuzman, A. Cohen, A. Zaks, and I. Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 327–337. IEEE, 2009.

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

M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

J. Xue. *Loop tiling for parallelism*, volume 575. Springer Science & Business Media, 2000.

M. V. Yason. The art of unpacking. *Retrieved Feb*, 12: 2008, 2007.

Q. Yi and K. Kennedy. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *The International Journal of High Performance Computing Applications*, 18(2):237–253, 2004.