UJOpt: Heuristic Approach for Applying Unroll-and-Jam Optimization and Loop Order Selection

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

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

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

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

Abstract

Loop transformations help exploit features such as parallelism (task and data-level parallelisms) and locality (temporal and spatial localities) to achieve higher performance in multi-dimensional loops. Loop unroll-and-jam is one such transformation that can exploit parallelism available in the outer loop. However, not all loops benefit from the unroll-and-jam transformation. Surprisingly this is true even when the unroll-and-jam transformation exposes fine-grain data-level parallelism and vectorization at non-innermost loops. In this work, we propose *UJOpt*, a heuristic based approach to determine whether unroll-and-jam is beneficial for a given loop nest and the best-performing loop order (in terms of lower execution cycles). Experimental evaluation on 32-core Intel Xeon Cascadelake architecture demonstrates that our approach identifies loop orders whose performance on average is within 7% from that of the *optimal* loop order, for loops taken from the Polybench test suite.

CCS Concepts

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

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725778

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. 2025. UJOpt: Heuristic Approach for Applying Unroll-and-Jam Optimization and Loop Order Selection. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3721145.3725778

1 Introduction

The continuous evolution in processor architecture over the last several decades has resulted in a single processor having multiple processor cores, each core supporting deep pipelines to enable pipelined parallelism, multi-threading to support concurrent execution of multiple threads, vector units to exploit fine-grained data-level parallelism, and multi-level cache hierarchy with sophisticated hardware prefetchers to enhance the memory bandwidth. Such a processor can exploit task-level parallelism using multiple cores, data-level (or SIMD) parallelism using vector processing units, thread, and instruction-level parallelisms. To take advantage of these architectural features and to extract higher performance, the applications running on these architectures should exploit these features effectively. Many compiler transformations/optimizations, specifically loop transformations [9], have been proposed to address this.

Loop tiling [37, 38] forms an important transformation on multi-dimensional loops. It divides the iteration space of a loop into small tiles that can exploit spatial and temporal locality of data accesses in loop nests. It maps a loop of depth n into loop nests of depth $2n$. The inner n loops, known as the intra-tile loops, run for a fixed number of iterations equal to

the tile size in that dimension. The outer n loops correspond to different tiles and are known as inter-tile loops. Loop permutation or interchange [3, 39] of a multi-dimensional nested loop is a loop transformation that reorders the iteration space of a loop by reordering its loop iterators. When loop tiling and interchanges are applied on n-dimensional loops, the $2n$ loop iterators in the loop can be permuted/reordered as permissible by the dependencies in the program.

Loop unroll-and-jam is another transformation technique [24] in which the second innermost loop[1] is unrolled by an unroll-factor value, and the resulting statements are jammed together. If the second innermost loop is parallel, but the innermost is not, then unroll-and-jam transformation helps to exploit this parallelism present in the second innermost loop. Existing loop transformation tools like Polly [17] and Pluto [12], which are based on Polyhedral techniques, are indeed quite useful in identifying and exposing different types of parallelism, data locality, and reuse. They also transform the given loop into a tiled loop with a valid loop order such that all the data dependencies present in the original loop are satisfied. They provide options to apply optimizations like loop unroll-and-jam . However, they fail to identify when to apply the unroll-and-jam transformation and what is the best loop order.

The performance of each valid loop order differs depending on how well the transformed code exploits task-, data-, and instruction-level parallelisms, to what extent it exploits temporal and spatial localities, how amenable are the accesses in the loop nest for vector load/store and data prefetching, etc. Among the many valid loop orders, the loop order with the lowest execution time/cycles is called the *optimal* loop order. In general, the interplay between these different architectural features and their impact on the performance of the transformed loop needs to be understood properly to generate efficient code for a given architecture.

The specific problem addressed in this paper is:

> Given an n-dimensional loop nest in which each of the loop dimensions is tiled, identify the loop order that results in the best performance when a combination of loop interchange and loop unroll-and-jam transformations are considered.

We explain this with the help of an example. Consider a 2-dimensional loop nest from `gemver` of Polybench [25, 26] suite as given in Listing 1.

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-iterations cannot be executed in parallel. Tiled versions of the loop nest with and without the unroll-and-jam transformation (with an unroll factor of 4) are shown below.

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

Listing 2: Tiled gemver_k2 loop

```
#pragma omp parallel
for(i=0;i<N/T;i++)
  for(j=0;j<N/T;j++)
    #pragma simd
    for(ii=0;ii<T;ii+=4)
      for(jj=0;jj<T;jj++)
        x[i*T+ii:ii+3]=x[i*T+ii:ii+3]+beta
                *A[j*T+jj][i*T+ii:ii+3]*y[j*T+jj];
```

Listing 3: gemver_k2 loop with unroll-and-jam transformation

Each version of the loop nest is fully permutable. There are six possible loop orders, $L1(i, j, ii, jj)$, $L2(i, j, jj, ii)$, $L3(i, ii, j, jj)$, $L4(j, i, ii, jj)$, $L5(j, i, jj, ii)$ and $L6(j, jj, i, ii)$ for each version. The dimensions ii and jj correspond to the intra-tile and i and j correspond to the inter-tile (We consider loop orders in which an intra-tile dimension appears within the corresponding inter-tile dimension.). The ordering of inter- and intra-tile dimensions dictate how the arrays (specifically 2-dimensional) are accessed, affecting how cache hierarchy and the hardware prefetchers are used. For example, in $L3(i, ii, j, jj)$, the dimensions j and jj traverse a row in each tile before moving on to the next row given by dimensions i and ii. An optimizing compiler can unroll the innermost loop multiple times in the tiled version (without unroll-and-jam), even though we do not show it explicitly. Such unrolling is mandatorily done when the innermost loop does not carry any dependency and can be vectorized.

For the above loop nest, without the unroll-and-jam transformation (shown in Listing 2), the innermost loop in the $L1(i, j, ii, jj)$ loop order cannot be vectorized (and hence cannot use the AVX instructions). On the other hand, the $L1(i, j, ii, jj)$ loop order with the unroll-and-jam version

[1] In fact, the unroll-and-jam transformation can choose any loops other than the innermost loop.

(shown in Listing 3) can exploit vector parallelism as the ii-dimension is unrolled and does not carry any dependency. This raises the question of how does $L1(i, j, ii, jj)$ loop order of the tiled version (without unroll-and-jam) performs compared to the $L1(i, j, ii, jj)$ loop order with unroll-and-jam. In general, the question that we try to answer is which among the 12 loop orders would perform the best. It can be observed that some loop order(s) may exploit vector parallelism, some may exploit task-level parallelism at the different loop levels (outermost, second, or third outermost loop), some loop orders may exploit spatial locality better, some loop orders are more amenable for data prefetching, etc. Depending on these, the performance of different loops may vary. Given this, can the compiler identify the *optimal* loop order (the one that results in the lowest execution cycles) for a given architecture?

In this work, we have developed an efficient heuristic to determine whether the unroll-and-jam optimization benefits a given loop nest and identify a best-performing loop order that is near-optimal. To the best of our knowledge, there is no work that considers a combination of loop tiling, loop interchange and unroll-and-jam together and identifies the best-performing loop order. We refer to our approach as *UJOpt*. Our approach uses the characteristics of a given loop nest, represented as high-level features, to decide whether unroll-and-jam is beneficial and the best-performing loop order. The proposed heuristic is evaluated on Intel Xeon Cascade Lake 8268 server using 32 cores. We evaluate our approach on loops from the Polybench [25, 26] test suite and a set of loops generated using a synthetic loop generator tool. *UJOpt* results in performance that is within 7% and 13% of the optimal performance for Polybench [25] and synthetic loops benchmark suites, respectively. The loop orders identified by *UJOpt*, achieve an improvement of 1.15x — 1.79x over Polly [17] and Pluto [12].

2 Background and Motivation

2.1 Background

2.1.1 Loop Transformation. Loop transformations refer to transforming a given multi-dimensional loop into a form such that the performance of the transformed loop is improved while retaining the correctness of its functionality. These transformations help exploit the underlying architectural features to realize higher performance for an application. Many loop transformation techniques exist, but we will limit ourselves to loop tiling, loop permutation, and unroll-and-jam.

Loop tiling [37, 38] divides the iteration space of the original loop into tiles/blocks to exploit temporal and spatial locality. The original loop nest with depth n is transformed into a loop nest with depth $2n$. The inner n loop iterators (intra-tile dimensions) run within a tile for a fixed number of iterations equal to the tile size. The outer n loop iterators (inter-tile dimensions) iterate across tiles covering the entire iteration space of a loop (see Listing 2). For a given loop, the tile size is important to exploit the spatial and temporal localities. The performance of the resulting transformed tiled loop is further enhanced if the outermost and the innermost loops are parallel and exploit coarse-grain data-level parallelism on multiple cores and fine-grain data-level parallelism on vector processing units, respectively.

Loop interchange/permutation [3, 39] refers to exchanging the order of loop iterators used by a nested loop. This transformation changes the order in which the iteration space of the original loop is traversed. The performance of the transformed loop depends on the levels at which coarse-grain and fine-grain parallelism are exploited and how effectively the multi-level cache hierarchy and the hardware prefetchers are used to extract higher memory bandwidth.

Listing 2 shows the tiled version of the loop given in Listing 1, as generated by Polly [17]. 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 interchanged/permuted such that the resulting order 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. In this work, we consider only 2-dimensional perfect loop nests. Hence, tiling and interchange give six different permutations of the loop nest. We denote them 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 $L6(j, jj, i, ii)$.

Loop unroll-and-jam transformation unrolls the outer loop multiple times (depending on the unroll factor) and jams or packs together the inner loop statements. This transformation is effective when parallelism is present in the outer loop than the inner loop. The transformed loop realizes high performance if it can exploit vector processing units, multi-level cache hierarchy, hardware prefetchers, and instruction level parallelism (ILP). For the loop shown in Listing 1, with i-dimension being parallel, the transformed unroll-and-jam code is shown in Listing 3. with an unroll factor of 4. In the resulting transformed loop, as the dimension i is parallel, the dimension ii will be vectorized.

The unroll-and-jam optimization can be applied to the above six loop orders. The performance of each loop order with unroll-and-jam again depends on how well it exploits multiple cores, vector processing units, caches, and hardware prefetchers. Hence, for a given perfect loop nest, there are twelve loop orders possible: six without unroll-and-jam and six with unroll-and-jam optimization. Our heuristics

Table 1: Normalized Execution Cycles for *gemver_k2*

	Normalized Execution Cycles for N=4096					
	L1	**L2**	**L3**	**L4**	**L5**	**L6**
without UJ	3.88	1.54	10.18	2.71	1.95	18.14
with UJ	**1.00**	1.16	3.39	1.85	1.71	4.81

take the characteristics of each of these loop orders and determine whether the application of unroll-and-jam optimization is beneficial and the loop order that gives the best performance.

2.2 Motivation

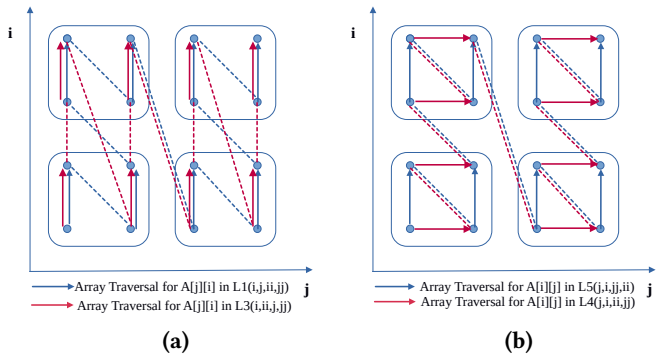
Next, we motivate our work using a few example loops taken from the Polybench benchmark suite.

2.2.1 Cases where unroll-and-jam is beneficial: Consider a loop from *gemver* benchmark as shown in Listing 1. The code after the tiling transformation is given in Listing 2. This loop is parallel in i-dimension, and the access pattern of the 2-D array A is of the form $A[j][i]$. Loop orders $L1(i, j, ii, jj)$, $L2(i, j, jj, ii)$, and $L3(i, ii, j, jj)$ exploit the coarse-level parallelism at the outermost loop on multiple cores, while for $L4(j, i, ii, jj)$, $L5(j, i, jj, ii)$, and $L6(j, jj, i, ii)$, the parallel threads need to synchronize for each j-iteration. Among $L1, L2, L3$, loop order $L2$ also exhibits fine-grain data-level (vector) parallelism, and as the array access pattern ($A[j][i]$) align with the intra-tile traversal it generates vector loads. On the other hand, $L1$ and $L3$ do not exploit vectorization, as the innermost dimension jj has loop-carried dependence and hence generate simple loads/stores.

The code generated after applying the unroll-and-jam transformation to *gemver_k2* loop is shown in Listing 3. For this transformed loop, out of the six loop orders, $L1(i, j, ii, jj)$, $L2(i, j, jj, ii)$, and $L3(i, ii, j, jj)$ exploit coarse-grain data-level parallelism. Among these three loop orders, $L1$ and $L3$ exploit fine-grain data-level parallelism (as ii dimension is parallel). For $L2$, the jj dimension is unrolled which is not parallel. Hence, it does not lead to any vectorization.

The normalized execution cycles of *gemver_k2* loop for an input size $N = 4096$ and a tile size of 32 with and without unroll-and-jam optimization 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 cycles) across all the 12 loop orders when executed on the Intel Xeon Cascadelake server.[2] Table 1 shows that this loop gets benefitted by the unroll-and-jam optimization with $L1$ as the best-performing loop order. Why does this particular loop order perform better?

[2]The details of our experimental framework and methodology are presented in Section 5.



(a) **(b)**

Figure 1: Array Traversal for Two Loop Orders

The loop statements generated for $L2(i, j, jj, ii)$ loop order without unroll-and-jam and $L1(i, j, ii, jj)$ and $L3(i, ii, j, jj)$ loop orders with unroll-and-jam exploit task- and data-level parallelism in a similar way. However, in loop orders $L1$ and $L3$ with unroll-and-jam , the writes to $x[ii]$ are optimized to scalar writes for the entire jj and j, jj iterations, respectively, and $x[ii]$ is updated only at the end of jj and j, jj iterations; whereas, in $L2$ loop order without unroll-and-jam , with ii being the innermost dimension, the array x is updated for every ii iteration. Thus, $L1, L3$ with unroll-and-jam perform fewer stores and hence are likely to perform better than $L2$ without unroll-and-jam . Further, among $L1$ and $L3$, with unroll-and-jam, spatial locality of 2-D array A is exploited in $L1$ as compared to $L3$ as shown in Figure 1a. The intra-tile dimensions of $L1$ traverse the current tile completely before moving to the next tile while in $L3$, the accesses are interleaved across tiles as shown in Figure 1a. Hence, $L1$ with unroll-and-jam is likely to be a better performing loop order. Here, even the best loop order without the unroll-and-jam transformation is 1.54x slower. Picking a wrong loop order can incur performance loss as high as 18.14x.

2.2.2 Cases where unroll-and-jam is not beneficial: Consider a loop from *adi* benchmark as shown in Listing 4. This loop has parallelism in j-dimension. All the arrays are 2-dimensional and have row-major ordering. Table 2 shows the normalized execution cycles for the *adi_k3* loop for input size $N = 8192$ and a tile size of 32 with and without the unroll-and-jam optimization.

```
for (i=1;i<N;i++)
 for (j=0;j<N;j++) {
  X[i][j]=X[i][j]-X[i-1][j]*A[i][j]/B[i-1][j];
  B[i][j]=B[i][j]-A[i][j]*A[i][j]/B[i-1][j];
 }
```

Listing 4: adi_k3 loop

Table 2: Normalized Execution Cycles for *adi_k*3

	Normalized Execution Cycles for N=8192					
	L1	**L2**	**L3**	**L4**	**L5**	**L6**
without UJ	1.13	2.49	3.39	**1.00**	1.86	8.49
with UJ	1.61	1.32	1.54	2.29	1.11	1.93

For this loop nest, as j-dimension is parallel, only loop orders $L4(j, i, ii, jj)$, $L5(j, i, jj, ii)$and $L6(j, jj, i, ii)$ exploit coarse-grain data-level parallelism (at the outermost level). Other loops orders will exploit coarse-grain parallelism where the inter-tile dimension of j occurs, and hence will involve synchronization for each iterations of the next outer loop level. Without unroll-and-jam optimization, among loop orders $L4, L5, L6$, only loop order $L4$ with the parallel jj-dimension at the innermost level, generates vector loads and stores, while $L5, L6$ do not vectorize the innermost loop and hence use simple loads and stores for all the 2-D arrays.

Table 2 presents the normalized execution time of this loop nest. From Table 2, it can be seen that this loop nest does not benefit from the unroll-and-jam transformation. Specifically, the performance of loop order $L4$ without unroll-and-jam is better than $L5$ with unroll-and-jam by 11%. Why doesn't unroll-and-jam help in this loop nest? Among $L5, L6$, loop order $L5$ exploits spatial locality more effectively than $L6$. Loop orders $L4$ without unroll-and-jam and $L5$ with unroll-and-jam exploit both coarse-grain and fine-grain data-level parallelism in a similar way. Further, both loop orders exploit spatial locality as they traverse the entire current tile before moving on to the next tile as shown in Figure 1b. However, the $L5$ traversal covers the tiles in a column first order, while the $L4$ order does it the other way around, resulting in a small performance difference. In our experiments, we found that there are loop nests where all loop orders with unroll-and-jam perform poorly compared to without unroll-and-jam.

3 Heuristics Based Best-Performing Loop Order for Unroll-and-Jam Optimization

In this section, we discuss our heuristic approach (*UJOpt*) to determine whether or not apply the unroll-and-jam transformation and the best-performing loop order for a given loop. We consider loop permutation, loop tiling and unroll-and-jam transformations. With these transformations, there are totally 12 different loop orders, six with and six without the unroll-and-jam optimization.

3.1 What Factors Influences the Performance of a Loop Nest?

The performance of a loop depends on how well it exploits different architectural features like multiple cores, vector units, cache hierarchy, hardware prefetchers. For the 2-D perfect loop nests considered in this work, 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. If the parallel dimension happens to be at the outermost level, the parallel threads can execute independently on multiple cores available without having to synchronize after each iteration. However, if the parallel dimension is one of the inner loops, then synchronization is required for each iteration of the next outer loop. Such synchronization overheads reduce the performance of the loop.

Similarly, if the parallel intra-tile dimension happens to be the innermost dimension, fine-grain data-level parallelism (vectorization) is exploited. Further, whether the parallel innermost dimension leads to vectorized loads/stores or scatter/gather accesses is determined by the access patterns of 2-D arrays. For example, the loop order $L2(i, j, jj, ii)$, with parallelelism in i-dimension, will lead to vector loads/stores of arrays with access pattern of type $A[jj][ii]$ whereas it generates scatter/gather accesses for arrays with access pattern $A[ii][jj]$. From a performance point of view, scatter/-gather accesses are more expensive than a vector load/store instruction. Last, the opportunity to exploit vectorization is lost, if the parallel intra-tile dimension happens to be one of the inner loops (but not the innermost loop).

Next, some of the loop orders lead to optimizing the read-/write accesses to arrays, if the array accesses do not include the index of the inner loop. For example, with loop order $L1(i, j, ii, jj)$ writes to $x[ii]$ are optimized to scalar writes (or writes to a register instead of to a memory location) for the entire jj-iteration, followed by write to memory $x[ii]$ only only at the end of jj-iteration. Similarly, loop order $L3(i, ii, j, jj)$ will replace the writes to $x[ii]$ for the entire (j, jj) iteration by writes to a temporary (register) followed by single memory written at the end of entire j-iteration. This operation where the writes to 1-D arrays are optimized reduce the cache writebacks and help in exploiting the multi-level caches more effectively.

Last, for a given set of array accesses, some loop orders exploit spatial locality better than others. More specifically, for a given array access pattern, some loop orders access all the elements in the current tile completely, before moving on to the next tile. In such traversals, all the cache lines of the current tile will be available in the L1-cache and such accesses do not incur any stalls. On the other hand, for the same array access pattern, some loop orders traverse the tiles in an interleaved fashion, accessing a single column from the current tile, all the way up to the entire iteration space and then accessing the next element from the first tile.

Such accesses experience cache misses as the cache line belonging to the current tile will be replaced by others, thus incurring stalls. For example, for array pattern type $A[j][i]$, loop order $L1(i, j, ii, jj)$ exploits spatial locality better than $L3(i, ii, j, jj)$ as shown in Figure 1a.

All these parameters influence the performance of a loop as they dictate how the architectural features will be exploited. Our heuristics take the characteristics of each loop order and decide the best-performing loop order based on the performance of a loop order with respect to these parameters. Developing an analytical model for estimating the cost (execution time) of different loop orders with and without unroll-and-jam would have helped to determine the best-performing loop order. However, our initial efforts to construct a performance model using linear regression did not yield good performance. Hence, we have proposed a simple yet practical heuristic that considers the characteristics exploited by the loop orders.

3.2 Heuristics for determining best-performing loop order

Each of the twelve loop orders considered with and without unroll-and-jam optimization are characterized by a few properties/features that distinguish them from each other. For each loop, we extract certain features such as dimensions along which parallelisms exist (i, j or both), the number of references to arrays of the form `A[f(i)][g(j)]`, `A[g(j)][f(i)]`, `X[f(i)]` and `X[g(j)]` in read and write operations, where `X` and `A` represent, respectively, one- and two-dimensional arrays, and `f` and `g` represent affine functions[3] of loop index variables `i` and `j` . A compiler can easily extract these features.

The chosen input features have bearing on the architecture and impact the performance of the loop nest. For example, the `A[f(i)][g(j)]` vs. `A[g(j)][f(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-dimension 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. Using these features, for each loop order, a vector representing its characteristics is generated. We consider the following characteristics for each loop order.

(1) A bit indicating whether the outermost loop is parallel.
(2) The number of references to 2-dimensional arrays that are vectorized.
(3) The number of references to 1-dimensional arrays that are vectorized.
(4) The number of references to 2-dimensional arrays with scatter/gather memory operations.
(5) The number of references to 1-dimensional arrays with write/store optimization.
(6) The spatial locality exploited on 2D (or higher dimensional) arrays. A value in the range of 1 to 6 (higher is better) is assigned to the 6 loop orders both with and without unroll-and-jam optimization, indicating the relative extent of spatial locality exploited by them.

[3]The constraints on the loop nest structure and affine accesses arise from the polyhedral frameworks.

Table 3: Characteristic Vectors for *gemver_k2* Loop

Loop Order	Loop Order Characters (1)	(2)	(3)	(4)	(5)	(6)
Without Unroll-and-Jam Optimization						
L1	1	0	0	0	1	5
L2	1	1	2	0	0	6
L3	1	0	0	0	1	4
L4	0	0	0	0	1	2
L5	0	1	2	0	0	3
L6	0	1	2	0	0	1
With Unroll-and-Jam Optimization						
L1	1	1	2	0	1	5
L2	1	0	0	0	0	6
L3	1	1	2	0	1	4
L4	0	1	2	0	1	2
L5	0	0	0	0	0	3
L6	0	0	0	0	0	1

Characteristics (2)-(4) of loop orders L1, L3, and L4 without unroll-and-jam and L2, L5, and L6 with unroll-and-jam will be the same, corresponding to the intra-tile dimension that gets vectorized and unrolled in them, and vice-versa. The characteristics numbered (1), (5) and (6) differ across loop orders depending on the parallel dimension, store operation and array access patterns in the loop order. Note that the input size (N) is not included as a characteristic of a loop order[4]. The loop order chosen by *UJOpt* will be the same across all N values of that loop nest. As we demonstrate in Section 6, we have used three different N values and our approach works well across different input sizes.

For the loop shown in Listing 1, the twelve characteristic vectors corresponding to twelve loop orders (first six without unroll-and-jam and next six with unroll-and-jam optimization) are as shown in Table 3.

Our heuristic algorithm is presented in Algorithm 1. It takes as input the above 6 characteristics of the 12 different loop orders, represented as a 12×6 matrix C and ST containing the list of all stores in the loop nest. From them,

[4]Our initial experiments with input size as a loop order characteristic did not result in significant benefits, affecting the performance for a few loops.

Algorithm 1: To Find the Best-Performing Loop Order with Unroll-and-Jam Optimization

```
1  Function Find_Best_Performing_Looporder(C,ST)
       Input: C[ℓ,k]: A 12 × 6 matrix, where ℓ
                represents the loop order and k
                represents the k^th characteristic
              ST: List of all stores in the loop nest
       Output: Best-Performing loop order
2      S1=Select from C the loop orders with parallel
         outermost dimension i.e., loop orders for which
         C[ℓ,1] is 1
3      if Majority of stores in ST are 2-D then
4          S2=Select loop orders from S1 that generate
             either zero or minimum number of
             scatters/gathers for 2-D arrays i.e., loop
             orders that have the lowest values for
             C[ℓ,4]
5          S3=Select loop orders from S2 that vectorize
             1-D or 2-D arrays i.e., loop orders for which
             C[ℓ,2] or C[ℓ,3] is 1
6      else
7          S2=Select loop orders from S1 that vectorize
             1-D or 2-D arrays i.e., loop orders for which
             C[ℓ,2] or C[ℓ,3] is 1
8          S3=Select loop orders from S2 that generate
             lower scatters/gathers for 2-D arrays i.e.,
             loop orders that have lower values for
             C[ℓ,4]
9      S4=Select loop orders from S3 that enable write
         optimizations i.e., loop orders for which C[ℓ,5]
         is 1
10     S5=Select loop order(s) from S4 that exploits
         spatial locality i.e., loop orders that have the
         highest value for C[ℓ,6]
11     if S5 contains only one loop order then
12         Output the loop order
13     else
14         Output the loop order without the
             unroll-and-jam optimization
```

only loop orders with parallelism present in the outermost level will be selected at line 2. In case of loops with both dimensions parallel, all 12 loop orders will be selected. For loops with either i- or j- dimension being parallel, only six loop orders (three each with and without unroll-and-jam) will be selected. The remaining loop orders which are not parallel in the outermost level will be discarded.

At line 3, a check is made whether the majority of store operations in ST contain 1-D or 2-D arrays. If the majority of the store operations contain 2-D arrays, then at line 4, the loop orders that generate either zero or minimum number of scatter/gather accesses for 2-D arrays are selected as below. Suppose all loop orders generate scatter-gather operations, which happens for loops with both parallel dimensions and 2-D arrays of both access patterns are present; then, a check is made whether there are loop orders with fewer scatters/gathers. If such loop orders are present then they are selected; else, all loop orders generate an equal number of scatters/gathers, and they will be selected. If not all loop orders generate scatters/gathers (which happens for either i- or j-dimension parallel loops or both dimension parallel loops with only one type of 2-D array access patterns), then only loop orders that do not generate scatters/gathers will be selected. From the selected loop orders, at line 5, only those loop orders that vectorize 1- and 2-D arrays will be selected.

If the majority of the store operations contain 1-D arrays, then at line 7, only those loop orders that vectorize either 1-D or 2-D arrays will be selected. At line 8, only those loop orders that generate zero scatters/gathers for 2-D arrays will be selected. In the next step, from the selected loop orders, only those loop orders that enable write optimizations will be chosen at line 9. At line 10, only loop orders that exploits spatial locality will be chosen among the loop orders selected in the previous step. At line 11, if the output set contains a single loop order, then it is chosen as the best-performing loop order as given by the heuristic. If there are more than one loop orders in the output set, then the loop order without unroll-and-jam is given as the output.

Intuitively, for loop nests with majority of 2-D store operations the heuristic does not recommend unroll-and-jam optimization, if the access pattern of all the 2-D arrays is same, as some loop orders without unroll-and-jam generate vector loads/stores of 2-D arrays while with unroll-and-jam there will be scatters/gathers. For loops with mixed 2-D array access patterns, unroll-and-jam optimization is recommended with the loop order that exploits spatial locality depending on the reference count of each access pattern. For loop nests with majority of 1-D store operations, if loops with parallel i- or j- dimension have all the 2-D arrays of the same access pattern (either `A[f(i)][g(j)]` or `A[g(j)][f(i)]`), then unroll-and-jam optimization is recommended. If 2-D arrays with both the access patterns are present, then unroll-and-jam is recommended if majority of the accesses are of the form `A[f(i)][g(j)]` for loops with parallel i-dimension and `A[g(j)][f(i)]` for loops with parallel j-dimension.

We remark here that UJOpt only chooses one among 12 loop orders based on the characteristic vectors of each loop orders. Checking the legality of loop transformation or enabling appropriate parallelization/vectorization is done by the Polly [17] framework, which ensures the correctness of our approach. Though, we use a simple heuristic approach

with a fewer characteristics of each loop order, it works well in practice as demonstrated in Section 6.

4 Implementation

We have implemented our tool *UJOpt* as a separate module for this work. Integrating the tools into mainstream compilers like LLVM with Polly [17] and GCC [36], requires marking the target loop using compiler directives and extracting the 12 characteristic vectors for the loopnest, which are then given as input to our *UJOpt* to identify the best loop order. The rest of the compiler tool chain can take the output of *UJOpt* and generate the code. The compilation process involves generating an appropriate jscop file based on the loop order selected by *UJOpt*. The generated jscop file along with appropriate set of compiler flags are then given to the LLVM compiler integrated with Polly [17]. The LLVM compiler tool chain generates the architecture specific target code, using the jscop file and compiler flags. This approach does not require any modification to the existing LLVM or Polly tool infrastructure. The overhead in terms of compilation time introduced by our method is relatively small (10's of milliseconds) and the compilation times are similar to the Polly [17] compiler. This small increase in the compilation time due to our heuristics could potentially result in a significant improvement in the execution time, making our heuristic approach favorable for real-world usage.

5 Experimental Methodology

5.1 Architecture Used

We have used an Intel Xeon Cascadelake 8268 system [6] for our experiments, with 48 cores in a 2-socket configuration. The base frequency of the processor is 2.9 GHz. The system has 192GB of DDR4 RAM operating at 2933 MHz. There are 2 memory controllers and 3 memory channels per controller. Each core has an L1d and L1i cache each of capacity 32KB per core and an L2 cache of capacity 1MB per core. It supports AVX-512 instructions. The cores in a socket share an L3 cache of capacity 35.75MB.

5.2 Measuring Execution Cycles

We measure the execution cycles for different loop orders of loop nests, with and without unroll-and-jam optimization, using the *_rdtsc* [2] function. We have used state-of-the-art compilers 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 generate the code for each of these versions. We use 32 cores of Intel Xeon Cascadelake server [6], to distribute the outer parallel loop evenly across all cores for the problem sizes we have considered. To minimize the variations in execution cycles, we had exclusive access to the server during the run, and hyperthreading on each core was disabled to minimize OS-related latencies[15, 29]. Further, to reduce the variations in the execution cycles across multiple executions of test 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 $1.5 \times InterQuartileRange(IQR)$ [30] 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 whole run is discarded if the CV is found to be 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.

5.3 Benchmarks Used

We have used twenty-three 2-dimensional perfect loop nests from Polybench-3.2 [25] 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) 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 4[5]. 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. We refer to this set of loops as Poly_Loops.

Table 4: 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		

These loops can be grouped based on the parallel dimension, type and access pattern of arrays of store operations, and the access patterns of the 2-D arrays present in the loop. To complete Poly_Loops set to contain loops from all the groups, we have included eight more synthetic loops in this test loop set. Thus, there are 31 loops in Poly_Loops test set. We have used three different problem size values, $N = 4096$,

[5]Note that test loops fdtd-2d and jacobi-2d have stencil-like computations.

Table 5: Performance Comparison

Test Set	Geometric Mean of Normalized Execution Cycles									
	UJOpt	Without unroll-and-jam			With unroll-and-jam			Oracle Predictor		
		SVM_Pred	Polly	Pluto	SVM_Pred	Polly	Pluto	SVM_Pred_Oracle	Polly_Oracle	Pluto_Oracle
Poly_Loops	1.07	1.23	1.92	1.54	1.24	1.34	1.48	1.16	1.30	1.25
Synth_Loops	1.13	1.56	1.68	1.33	1.22	1.30	1.44	1.19	1.19	1.14

$N = 6144$, and $N = 8192$, in our evaluation[6]. We fixed the tile size to 32 and unroll factor to 4. We obtain the execution cycles for all the twelve loop orders and for all the problem sizes of a test loop on Intel Xeon Cascadelake server and the loop order with the minimum execution cycles is the best-performing one. We also evaluate the performance of our heuristics on another set of test loops obtained using a synthetic loop generator tool, which generates 2-D loop nests whose characteristics (as listed in Section 3.2) match that of real-world loops. We generate 100 synthetic loops of each problem size N in this set and we refer to this test set as Synth_Loops.

6 Results and Discussions

This section discusses the results of our heuristics-based approach for loop nests on Intel Xeon Cascadelake server. We refer to our proposed heuristics-based approach as *UJOpt*. We compare the execution cycles obtained using our approach with an *oracle* method (referred to as *Optimal*) that always picks the best-performing loop order among the 12 loop orders. We report the normalized execution cycles of our approach, normalized with respect to the *Optimal* approach. This is a "lower is better metric" and indicates how far the performance of the identified loop order of the method is from the *Optimal* 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.

We compare the performance of *UJOpt* with (i) the code generated by Polly [17] and Pluto [12] and (ii) SVM_Pred [8], a highly-tuned Support Vector Machine (SVM) based method that predicts the best-performing loop order with and without the unroll-and-jam optimization. We do not consider production compilers like GCC [36] and Clang [21] as they do not tile or apply a loop order automatically, and require the end user to manually tile the loop and select a loop order. However, Polly [17] and Pluto [12] which we consider for evaluation project are somewhat similar comparison as Polly [17] uses the LLVM infrastructure underneath (similar to Clang [21]. Compilers like TVM or AutoTVM[13] used for deep learning optimizations, consider only tiling and loop order selection but do not consider loop unroll-and-jam transformation.

Further, for each of these methods, we report the performance of an (infeasible) oracle version of each of these methods that always knows whether or not to apply the unroll-and-jam transformation for a given loop nest. In all the methods, we used a fixed tile size of 32.

6.1 Performance Comparison

Table 5 reports the geometric mean of normalized execution cycles of different methods, normalized with respect to the oracle *Optimal* method. Our approach *UJOpt* performs better than all the compared methods for both the test sets, achieving performance within 7% and 13% from the *Optimal* loop order. *UJOpt* outperforms Polly and Pluto by a significant margin: *UJOpt* achieves 1.48x – 1.79x performance improvement[7] over Polly without the unroll-and-jam optimization and 1.15x – 1.25x over Polly with the unroll-and-jam optimization. Performance comparison over Pluto numbers range from 1.18x – 1.44x and 1.27x – 1.38x without and with the unroll-and-jam optimization. *UJOpt* also outperforms SVM_Pred [8] which is an SVM based loop order predictor by 1.15x – 1.38x and 1.08x – 1.16x, respectively, without and with the unroll-and-jam optimization. With our rigorous execution cycle measurement procedure discussed in Section 5.2, the performance benefits reported in our results are significantly higher than the 2% CV that we used in the measurement.

Interestingly, our heuristic approach *UJOpt* outperforms the competing methods even when we assume that they (Polly, Pluto and SVM_Pred) have an *oracle* capability to decide on whether or not to apply the unroll-and-jam optimization. The performance improvement of *UJOpt* over these methods (with the *oracle* capability) range from 16% – 30% for Polly loops and 14% – 19% for synthetic loops. At best, the performance of Pluto_Oracle is close to that of *UJOpt* for Synth_Loops. However, oracle predictors are hypothetical and infeasible.

[6]These input sizes result in working set sizes which are beyond large L3 cache (35.75 MB of Intel Xeon Cascadelake processor.)

[7]These performance numbers against other competitive methods are obtained by taking the ratio of normalized execution cycles of Polly to that of *UJOpt* method.

The average prediction accuracy, i.e., the ratio of the number of times *UJOpt* identifies the best-performing loop order (among the 12 loops orders) to the total number of loops in the benchmark is 70% and 52% for Poly_Loops and Synth_Loops, respectively. Although the prediction accuracy values are not significantly high, the average performance loss values of *UJOpt* over *Optimal* are only 7% and 13% for the two benchmark sets. This is because even when our heuristic did not pick the best-performing loop order, it selects either the second or third best-performing loop order. To capture this and to give a better measure of accuracy, we define top-2 accuracy metric, which considers the prediction is accurate if the selected loop order is the lowest or the second lowest. Similarly, for the top-3 accuracy metric, the prediction is considered accurate if our heuristic selected a loop order which is within top-3 configurations among the twelve loop orders. The top-2 and top-3 prediction accuracies for our methods are 88% and 95%, respectively.

Polyhedral frameworks like Pluto [12] and Polly [17] use simple heuristics to identify the loop order. Polly without unroll-and-jam incurs significant losses, as it does not exploit data-level parallelism using vectorization in certain loop nests. Pluto [12], on the other hand, optimizes for vectorization and therefore performs slightly better than Polly. However, both methods (with and without the unroll-and-jam optimization) still perform poor compared to *UJOpt*. Both, Polly and Pluto determine the same loop orders for a loop with and without unroll-and-jam optimization, without taking into consideration that with unroll-and-jam optimization, the second innermost intra-tile dimension gets unrolled.

The performance losses incurred by Polly with unroll-and-jam are lesser than those without the unroll-and-jam optimization. Polly gives the same loop order with and without unroll-and-jam optimization. For most of the loops with parallel i-dimension, with unroll-and-jam optimization, the loop orders given by Polly unroll the parallel intra-tile dimension, leading to vectorized accesses of 1- and 2-D arrays, and are the best-performing loop orders, thus reducing the performance losses incurred. Pluto, on the other hand, performs no additional optimizations with unroll-and-jam other than unrolling the second innermost loop dimension, although it takes vectorization into account.

6.2 Complete Benchmark Execution

Here, we consider five representative benchmarks from the Polybench-3.2 suite [25, 26] having more than one kernel, and measure the execution cycles taken for the complete execution of the benchmark for each problem size. For Polly [17] and Pluto [12], we consider the cases with and without unroll-and-jam optimization separately, while for our approach, we consider the loop orders given by our heuristic model *UJOpt*. Figure 2 shows the normalized execution cycles, normalized with respect to Polly_UJ, for all the five methods (Polly_NoUJ, Pluto_NoUJ, Polly_UJ, Pluto_UJ and UJOpt), for problem size 8192. It can be observed that *UJOpt* performs better than all the other four methods across all the five benchmarks. We observe similar trends for problem sizes 4096 and 6144. Overall *UJOpt* outperforms Polly and Pluto without unroll-and-jam optimization by 48.34% and 14.2%, respectively, and Polly and Pluto with unroll-and-jam optimization by 47.36% and 21.48%, respectively.

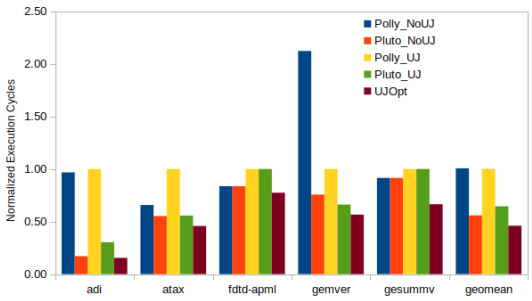


Figure 2: Complete Benchmark Execution Performance

6.3 Handling Higher Dimensional Loops

In this work, so far, we have considered only two-dimensional perfect loop nests. When the dimensionality of the loop nest is increased beyond, the number of legal orders to be considered (with or without the unroll-and-jam optimization) increases significantly. For example, for a 3-dimensional loop, the number of loop orders with unroll-and-jam is 180. Extending our heuristic approach to such higher dimensional loop nests involves dealing with a large number of loop orders, and this may in turn reduce the prediction accuracy and the effectiveness. Instead, we propose extending our approach by applying the heuristic only to the innermost two loop orders.

We decide the innermost two loop orders to be used, by 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 [25, 26] 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 *UJOpt* is within 16% of the *Optimal* loop order

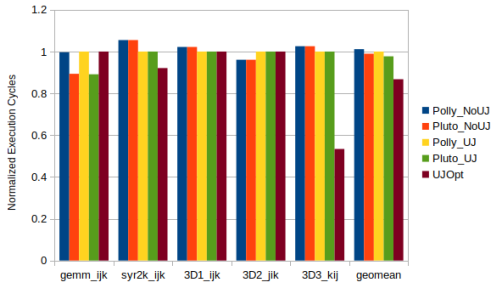


Figure 3: Executing Higher Dimensional Loops with *UJOpt*

among the 36 loop orders with or without the unroll-and-jam (12 loop orders for each of the three loop permutations), as shown in Figure 3. Note that for 3D3_kij loop, *UJOpt* performs significantly better than all other methods. This is because the loop orders selected by other methods do not exploit spatial locality for some arrays, where as *UJOpt* identifies a loop order that enables vectorization and exploits spatial locality, thus performing better than other methods.This demonstrates that our approach can be extended to higher dimensional loops.

6.4 Sensitivity Study

6.4.1 Changing Core Count and Tile Size. For the experiments discussed so far, we have used 32 (out of 48) cores of the Intel Xeon Cascadelake processor [6] and a fixed tile size of 32. To understand the performance of *UJOpt* for different number of cores and tile size, we experimented by varying these parameters. The results of these experiments are presented in Table 6.

On Intel Xeon Cascadelake processor we used two different core counts, 16 and 48, keeping the tile size as 32. The geometric mean of normalized execution cycles across the 93 benchmark loops (31 loop nests (23 Polybench loops + 8 synthetic loops) of three input sizes) for each core count is presented in Table 6. Note that normalization of execution cycles for each loop nest and core count is done with respect to the oracle *Optimal* loop order of that loop nest for the given core count. We find the performance of *UJOpt* across different core counts to be somewhat similar — less than 9% performance loss compared to the oracle *Optimal* method. Similarly, when the tile size is changed to 64 (with core count of 32), the geometric mean of normalized execution cycles across the 93 benchmark loops for *UJOpt* is 1.07, demonstrating that the performance of our approach is within 7% of the *Optimal*. These experiments indicate that the heuristic used in *UJOpt* performs well across changing core count and tile size parameters.

Table 6: Performance of *UJOpt* for Different Core Counts and Tile Sizes

Intel Xeon Cascadelake		
Tile Size	**No. of Cores**	**GeoMean of Normalized Execution Cycles**
32	16	1.09
32	32	1.07
32	48	1.09
32	32	1.07
64	32	1.07
AMD EPYC 9654 Processor		
32	8 (1 NUMA)	1.12
32	16 (1 NUMA)	1.13
32	32 (2 NUMA)	1.13
32	32 (4 NUMA)	1.14

Further, we have also evaluated *UJOpt* on AMD EPYC 9654 processor [10], which is the fourth generation processor of the AMD EPYC server processor family. The processor cores can be organized into one, two or four NUMA domains [4, 10]. On this processor, we fixed the tile size as 32 and experimented using three core count values, 8, 16 and 32 under different NUMA settings. The geometric mean of normalized execution cycles of the loop order identified by *UJOpt* across different core counts and NUMA configuration varied between 1.12 –1.14. Note again that the normalization is with respect to the oracle *Optimal* loop order for the respective configuration. Thus, the performance of *UJOpt* was within 12% – 14% from *Optimal* across all these experiments. These experiments indicate that the characteristic vectors considered by our heuristic capture the loop order characteristics effectively and our approach works well across different target architectures.

6.4.2 Changing Unroll Factor. The experiments discussed so far use an unroll factor of 4. We experimented increasing the unroll factor to 8 for 10 Poly_Loops kernels. Our heuristic works well when the unroll factor is set to 8, although the performance benefit decreases slightly as compared to the unroll factor of 4. *UJOpt* performs within 12% of *Optimal* for 30 Poly_Loops kernels (10 kernels each with 3 different input sizes) with an unroll factor of 8. It achieves an improvement of 1.69x and 1.20x over Polly and Pluto (without unroll-and-jam) and 1.20x and 1.53x over Polly and Pluto (with unroll-and-jam). We observe that about 23% of the loops do not get benefited by increased unroll factor with unroll-and-jam and hence contribute towards the observed performance loss.

6.5 Using Machine Learning Approaches

Can we apply machine learning techniques to identify the best-performing loop order with and without the unroll-and-jam optimization? This can be solved as a classification

problem by applying the supervised machine learning techniques. The best-performing loop order of a loop (a number in the range of 1 to 12) is the output class. The output class for the loop nests are labeled by selecting the loop order that results in the lowest execution cycles. We experiment using Support Vector Machine (SVM) [14, 23] supervised machine learning techniques to this labeled data. We use the synthetic loops set, Synth_Loops, as the training data set and Poly_Loops as the test set. The train and test data sets are characterized by features (discussed in Section 3.2, such as the parallel dimension (categorical variables), and the number of references to arrays of the form `A[f(i)][g(j)]`, `A[g(j)][f(i)]`, `X[f(i)]`, and `X[g(j)]`, where `X` and `A` represent one- and two-dimensional arrays, and `f` and `g` represent affine functions of loop index variables `i` and `j`. Additionally, we included a few other features such as the problem size N and the number and the type of store operations. The train and test data were normalized to zero mean and unit variance for all our experiments.

6.5.1 C5.0 Classification Tree. We have trained a classification tree based on C5.0 decision tree algorithm. We have used the tree-based C5.0 implementation from the C50 [20] package. The hyperparameter "trials", which is the number specifying the boosting iterations was varied between 5 to 50, and we report the best performance across all iterations. The geometric mean of normalized execution cycles of the predictions made by this classification tree over *Optimal* was 1.42 with a prediction accuracy of 26.88%. Although classification tree methods have the advantage of explainable behavior, the higher performance loss and lower prediction accuracy discourage us from pursuing it further.

6.5.2 SVM Classifier. We have used the SVM implementation `e1071` [22], 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. For our experiments, we have explored both linear and radial basis function kernels of the e1071 package of R(version 3.6.0). However, SVMs with linear kernels performed better on our data set. The regularization hyperparameters in the objective function of the SVM classification problem were tuned using the cross-validation technique. These hyperparameters varied over 2^{-5} to 2^{5}. The hyperparameter values, which resulted in the best validation set performance, were used for final training and prediction purposes. We have used 5-fold cross-validation to avoid selection bias in the model. The geometric mean of normalized execution cycles of the predictions made by this SVM classifier model over *Optimal* was 1.27 with a prediction accuracy of 46.24%. The performance of this SVM classifier is no better than our *UJOpt*; however it performs better than Polly and Pluto.

Table 7: Performance of Machine Learning Techniques

	Geo Mean of Normalized Execn. Cycles			
Test Set	**C5.0**	**SVM**	**SVM_Tuned_100**	**SVM_Tuned_200**
Poly_Loops	1.42	1.27	1.22	1.17

We further tune the hyperparameters of the SVM radial kernels using grid search methods to build a more robust SVM classifier referred to as SVM_Tuned. The performance loss incurred by SVM_Tuned over *Optimal* is 1.22 with a prediction accuracy of 48.39%. The performance of the machine learning techniques are summarized in Table 7. It can be observed that, in our experiments, machine learning techniques do not perform as well as our heuristic approach *UJOpt*. The main reason for the sub-optimal performance of machine-learning techniques could be that the training data set used for building these models is a is small and contains only 100 loops. It may not cover the entire set of all the output classes, which could be one of the main reasons for the average performance of these machine learning models. By increasing the training data set to 200 loops, the performance loss over *Optimal* was reduced from 1.22 to 1.17.

We have provided the machine-learning models as a reasonable alternative to heuristic and our initial machine-learning models are based on the existing features of the training data sets. Improving the performance of these models can involve systematic exploration of the training space or considering other options beyond C5.0 and SVM. By capturing representative input features and using an extensive training data set, these machine learning approaches can be extended to build more robust models. However, such an approach would also require larger training overheads. In contrast, our *UJOpt* uses simple heuristics (based on domain knowledge) and is effective on both benchmark sets.

7 Related Work

Most of the work on loop transformations is towards coming up with a compiler framework or use machine learning (ML) based approaches to predict to explore the sequence of loop transformations for the underlying architecture. Stock et al. [34] develop an ML model trained using tensor contractions and consider loop permutation, vectorization and unroll-and-jam transformations. However, they do not consider exploiting coarse-grain parallelism along with vectorization. In our work, the heuristic selects loop permutation with and without unroll-and-jam optimization, that can en-

able both parallization and vectorization. Our heuristic selects loop orders taking to account synchronization overheads that may be incurred at the outer loop level (depending on the dependencies and the chosen loop order). Further, our approach handles loops with parallelism in either or both the dimensions, while they consider only loop nests with parallelism in *all* dimensions. Further, our approach extracts the features of the loop nest without generating the object code for each loop order or variant, in order to generate the characteristic vectors for them. This could be an important factor in the production compilers. While their approach can handle multi-dimensional loops but does not take tiling into account. Our approach considers tiling but is limited to 2-dimensional loop nest, with extension to multi-dimension loop nest in a limited setting.

Girbal et al. [16] present a framework based on 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. Trifunovic et al. [35] present a fast and accurate cost model and a framework to extract vectorization opportunities using polyhedral representation. Pouchet et al. [28] propose the decomposition of the optimization problem represented as convex polyhedron into sub-problems of much lower complexity, introducing *fusibility* concept in PoCC [27]. Bondhugula et al. [11] 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 [19] using *codelets*.

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 [12]. Grosser et al.[17] implement polyhedral techniques on top of the LLVM framework to transform and optimize parts of the program in a language-independent way. While [17] 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 [12] and [17]. The work in [31] looks at selecting unroll factors for perfectly nested loops automatically and generating the compact code for the selected unroll factors. We focus on when to apply the unroll-and-jam optimization, which could benefit by using a best-performing unroll factor. To the best of our knowledge, there does not exist any work that integrates loop tiling, loop permutation and loop unroll-and-jam transformations using heuristics to exploit coarse-grain parallelization, vectorization, spatial locality and write optimization. We also show substantial improvements over existing tools.

In [1], the authors develop an ML model by training a set of programs for loop unrolling, common subexpression elimination, if hoisting, and copy propagation. Ashouri et al. [7] apply ML techniques to model and predict the phase order of compiler optimization sequences Haj-Ali et al. [18] use different ML methods for an auto-vectorization method, but focus only on vectorization. Stephenson et al. [33] apply supervised classification to predict the unroll factors for loops. OpenTuner [5] ensembles different search techniques for autotuning a code for a given architecture based on the architectural parameters. Sioutas et al.[32] developed an analytical model for Halide DSL to select the cache hierarchy level and the tile size such that cache misses are reduced. These focus more on memory hierarchy and domain-specific languages.

8 Conclusion

In this work, we proposed a heuristic based approach for identifying the best-performing loop order for a given loop nest with unroll-and-jam optimization for multicore architectures. Our proposed approach considers loop permutations with unroll-and-jam that can enable parallelization and vectorization and identifies the best-performing loop order based on the characteristics of different loop orders. Our proposed heuristic approach identifies loop orders that are within 7% and 13% of the *optimal* loop order for the two benchmark sets studied and outperforms state-of-the-art techniques, Pluto and Polly.

Acknowledgments

We acknowledge the research fundings received from Intel Technology India Private Limited and Volvo Group India Private Limited which supported this work.

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, 11–pp.

[2] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. 2010. Breakthrough AES performance with intel AES new instructions. *White paper, June* 12 (2010), 217.

[3] John R Allen and Ken Kennedy. 1984. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*. 233–246.

[4] AMD. 2024. *4th Gen AMD Processor Architecture*. Technical Report. AMD.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Ravi Bhargava and Kai Troester. 2024. AMD Next Generation" Zen 4" Core and 4 th Gen AMD EPYC™ Server CPUs. *IEEE Micro* (2024).

[11] 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.

[12] 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.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Max Kuhn, Steve Weston, Mark Culp, Nathan Coulter, Ross Quinlan, et al. 2015. Package 'C50'. *CRAN, UTC* (2015).

[21] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5. 1–20.

[22] 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).

[23] David Meyer and FT Wien. 2015. Support vector machines. *The Interface to libsvm in package e1071* 28 (2015), 20.

[24] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short simd architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2–11.

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] Vivek Sarkar. 2000. Optimized unrolling of nested loops. In *Proceedings of the 14th international conference on Supercomputing*. 153–166.

[32] 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.

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

[34] 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.

[35] 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.

[36] William Von Hagen. 2011. *The definitive guide to GCC*. Apress.

[37] Michael Wolfe. 1987. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. 357–361.

[38] Jingling Xue. 2000. *Loop tiling for parallelism*. Vol. 575. Springer Science & Business Media.

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