A Deep Learning Model for Loop Interchange

Lina Mezdour[*]
hl_mezdour@esi.dz
NYU Abu Dhabi
ESI

Khadidja Kadem[*]
hk_kadem@esi.dz
NYU Abu Dhabi
ESI

Massinissa Merouani
mm12191@nyu.edu
NYU Abu Dhabi

Amina Selma Haichour
a_haichour@esi.dz
ESI

Saman Amarasinghe
saman@csail.mit.edu
MIT

Riyadh Baghdadi
baghdadi@nyu.edu
NYU Abu Dhabi

Abstract

Loop interchange is an important code optimization that improves data locality and extracts parallelism. While previous research in compilers has tried to automate the selection of which loops to interchange, existing methods have an important limitation. They use less precise machine models. This is mainly because developing a model to predict whether to interchange two loops is challenging since such a prediction depends on many factors. While state-of-the-art methods try to avoid this problem by using a deep-learning based cost model, they suffer from another limitation. They scale proportionally with the number of loop levels of a given loop nest. This is mainly because they use the model to evaluate all the possible loop interchanges (or a subset of the most promising ones). In this paper, we propose a novel deep-learning model for loop interchange that addresses the previous limitations. It takes a code representation as input and predicts the best pair of loops to interchange. Compared to state-of-the-art deep-learning based cost models, it requires constant time to predict the best loop interchange. This is in contrast to state-of-the-art deep learning models that are used to evaluate all the loop pairs and then pick the best one. The proposed model is the first deep learning model that requires a constant time to predict the best loops to interchange. The model is implemented and evaluated in the Tiramisu compiler, a state-of-the-art polyhedral compiler. We evaluate the proposed model on a benchmark of Tiramisu programs and show an accuracy of 76.66% for 1-shot and 94% for 2-shots. Experiments show that our model outperforms the cost model currently used by the Tiramisu compiler by 6.66% in terms of 1-shot accuracy, and 14% with 2-shots accuracy, while at the same time reducing the total execution time needed for predicting the best pair of loops to interchange.

Keywords: loop interchange, automatic code optimization, cost model, deep learning, compilers, Tiramisu

1 Introduction

With the increasing need for high performance code and the increasing complexity in hardware, code optimization is becoming more and more important. An example of important code optimizations is loop interchange (a.k.a., loop permutation or loop reordering). It is an optimization that aims at interchanging, or permuting, two loop levels with the goal of improving data locality or enabling parallelism.

Different compilers use different methods to choose the best loops to interchange. Some use analytical models [8, 15]. However, such models are less precise and do not necessarily take into consideration all the factors to correctly optimize code. To illustrate this, let us look at the following code of an image processing convolution (Figure 1). When this code is optimized by Pluto[8], a state-of-the-art polyhedral compiler, the resulting code is 17 times slower than hand-written code optimized for a 24-core machine[1].

```
for c in 0..3
  for x in 1..WIDTH-1
    for y in 1..HEIGHT-1
       output[c,x,y] =
       (img[c,x+1,y-1] + img[c,x+1,y] + img[c,x+1,y+1] +
        img[c,x  ,y-1] + img[c,x  ,y] + img[c,x,  y+1] +
        img[c,x-1,y-1] + img[c,x-1,y] + img[c,x-1,y+1])/9
```

Figure 1. Image processing convolution.

The main reason for the slowdown is that Pluto uses a linear cost model that does not capture the full complexity of the space of code optimizations. More precisely, Pluto uses Integer Linear Programming (ILP) to minimize the distance between producer and consumer statements. While this objective maximizes data locality and outermost parallelism, considering only these two factors is not enough. In this code, Pluto decides to parallelize the outermost loop, however, because the extent of this loop is three, only three threads can be run in parallel, which under-utilizes the 24-core machine being used. A better optimization approach would be to interchange the first two loops, and then parallelize the new outermost loop. Although this reduces data locality (since accesses to *img* would become non-contiguous), the resulting program would run faster as it is now able to utilize many more parallel threads.

[*]Both authors contributed equally to this research.

[1]1080p input image

Recent research has tried to address the previous problem by building deep-learning models to find the best loop transformations (including loop interchange)[2, 5, 9]. For instance, Tiramisu[5], another state-of-the-art polyhedral compiler, uses a deep learning cost model to predict the best loop interchange to apply. It applies loop interchange on each pair of loops in the code, and uses a deep-learning based cost model to predict the speedup in each case. It then picks the loop interchange that provides the best speedup. While this approach improves the accuracy of the model, the time needed to find the best loops to interchange grows quadratically with the number of loop levels, making the search space of the loop interchange transformation large. In addition, as each possible loop interchange is combined with the exploration of further code optimizations afterwards (loop parallelization, skewing, reversal, tiling, unrolling, etc.), the search space becomes much larger.

To overcome these limitations, we propose a novel deep learning model that only requires constant time to predict the best loops to interchange. It takes, as input, a loop nest representation and predicts, as output, the best loops to interchange (if interchange is needed). In comparison with state-of-the-art deep-learning based cost models, it predicts directly which loops should be interchanged, and therefore it only requires constant time to find the best loops to interchange, instead of requiring quadratic time, which reduces the overall code optimization time. The proposed model is designed for Intel Xeon E5-2695 CPUs (12-cores). It is integrated in the Tiramisu compiler, and is trained on a dataset of randomly generated Tiramisu programs. Each program in the dataset is labeled with the best loop interchanges.

While our model is specific to a particular CPU (on which data was collected), our approach itself is hardware-independent and can be reproduced for other CPU architectures without any adaptations being required. This ease of portability is due to the fact that the model's architecture, and the input characterization adopted are hardware independent. To adapt the model to other architectures, one need only to generate data on that machine and then retrain the model.

The contributions of this paper are as follows:

- We present the first deep learning model for predicting the best loops to interchange in constant time.
- We release a dataset of 208919 Tiramisu programs, each labeled with their best loop interchange.
- We integrate the proposed cost model into the auto-scheduler of Tiramisu and provide and evaluation compared to state-of-the-art polyhedral compilers.

In the first section of this paper, we present a background on loop interchange. Then, we present the proposed model, the representation of its input and the dataset used in the training. Last, we evaluate the model on standard benchmarks and synthetic programs.

2 Background on Loop Interchange

Loop interchange is a high-level code optimization. It operates by permuting two loops in a loop nest, in order to improve parallelism and data locality [3].

An example of why loop interchange is important is shown in the following example (Figure 2). C language is used in the example so the 2D-arrays are stored in row-major order. We suppose that the cache memory line size is enough to load a whole row of the array **A**.

```
for ( j = 0, j < M, j++ )
    for ( i = 0, i < N, i++ )
        A[i,j+1] = A[i,j] + 1;
```

Figure 2. Example of a simple 2-loop program

For each two successive iterations (j,i), the memory accesses of one iteration belong to a different cache line than the previous one. This may result in a cache miss that produces an extra access to the main memory to fetch the new line. Such accesses cause performance degradation, as $N * M$ main memory access will be needed.

However, using loop interchange, we can permute the i and the j loops, resulting in the following code (Figure 3).

```
for ( i = 0, i < N, i++ )
    for ( j = 0, j < M, j++ )
        A[i,j+1] = A[i,j] + 1;
```

Figure 3. Example of a simple 2-loop program after loop interchange.

This way, for each iteration of the outermost loop i, all the memory accesses are within the same cache line, which means that less memory accesses are needed (only N access at most) which improves the performances. Furthermore, this new order makes the outermost loop iterations independent of each other, which allows their parallelization improving performance further.

Let us take another example to further show the effect of loop interchange. Let us take the matrix multiplication algorithm. In this example, we measure the execution time for all the possible loop interchanges in the code (i.e., all the possible loop orders). The speedup of the transformed program over the original program, measured after application of each of the possible loop interchanges, is shown in Table 1 (in the table, I(L0,L1) means interchanging the loop level 0, which is the outermost, with loop level 1).

We can see that interchanging the two inner loops has quadrupled the performance of the code. However, interchanging the inner- and the outer-most loops made the performance nearly 3 times worse.

Table 1. Speedups obtained after the application of different loop interchanges on matrix multiplication.

Loop interchange	Measured Speedup
None	1.00
I(L0,L1)	0.90
I(L0,L2)	0.37
I(L1,L2)	4.04

3 Dataset Construction

Since our goal is to build a deep learning model for choosing the best loop interchanges to apply on an input program written in Tiramisu, we need a dataset containing Tiramisu programs to train the model on. Furthermore, since our model will be trained in a supervised manner, we need to label the dataset, by defining the best loop interchanges for each program.

Tiramisu [6] is a domain-specific language (DSL) embedded in C++. It provides a C++ API that allows users to write a high level, architecture-independent algorithm and a set of API calls to select which code transformations should be applied. It uses the polyhedral model internally [4, 6, 8, 12], to represent code, code transformations, and to reason about the correctness of code transformations. A Tiramisu program has two parts: an algorithm, describing the computation to-be-executed, and a schedule, describing in which order code should be executed (i.e., how it should be optimized). This separation allows an easier switching between different schedules.

However, in order to build a dataset, a large number of Tiramisu programs are required. To solve this challenge, Baghdadi et al [5] have resorted to generating synthetic data in order to train their cost model, used currently in the Tiramisu auto-scheduler. These programs are generated automatically, simulating real programs by using probabilistic distributions. Each one of these randomly generated programs are coupled with different schedules generated randomly, then compiled and executed, and finally, a measure of the execution time is recorded. The final dataset is composed of the tuples, where each tuple is a program, a schedule, and an execution time.

We used the same methodology described above to generate the dataset needed to train our model. We generated 208919 data points, each data point is a pair of a program, and a list of loop interchanges ordered from the best to the worst according to their recorded execution time. The machine used to generate the data is described in Section 6.

4 Program Characterization

The input of the model is represented as a set of simple high-level features, extracted from the code. This way, neither the compilation nor the execution of the program are needed to obtain the features used by our model. This representation has also been used by the cost model of Tiramisu [5]. It is based on the AST (Abstract Syntax Tree) representation of programs. Every leaf represents a statement (computation [6]). Every non-leaf node represents a loop level, with the root being the outermost loop in the program.

A detailed example of the input representation is illustrated in the Figure 4. The program used in the example contains 2 instructions (2 computations in Tiramisu's terms), whose depth are 3 and 4 respectively. The two outermost loops are shared between them (the program structure is represented by its AST in the figure). Each instruction is represented by a computation vector. This structure is composed of two sub-vectors: loop nest representation vector and assignment vector. Their content is described below.

- **Loop nest representation vector**: This sub-vector contains information about the loop levels surrounding the computation being represented. Every entry in the vector represents a loop, and thus is composed of: the upper and lower bounds of the loop, as well as two boolean tags, one expressing whether loop fusion has been applied on the loop, and the other is set if the loop level dimension is contributing in defining the computation output buffer's dimension.
- **Assignment vector**: This records the array accesses used in both the left- and right-hand sides of the statement. Each one is presented by an access matrix and an identifier. The former stores the coefficient of the array access, in the same way as the polyhedral model representation for array accesses [13]. Each row represents a buffer dimension, and each column a loop iterator (to which we add an extra column for constants). Only affine array accesses are represented by this schema, as Tiramisu being polyhedral, only supports affine accesses [5]. The assignment vector also includes integers representing the total count of the operations in the assignment (addition, multiplication, division, and subtractions). Last, an extra integer is added that contains the loop nest depth of the computation.

5 Model's Architecture

The proposed model aims to predict the best loops to be interchanged in a Tiramisu program (if any loop should be interchanged). Since the set of loop pairs in a program is discrete, we represent this problem as a classification problem.

The architecture of the proposed model is similar to that used by Baghdadi et al. [5]. In the latter, the first two layers are responsible for extracting a vector representing the Tiramisu program including all of the loops and computations of that program. In our work, we have used the architecture of these first two layers, and added a last classification layer.

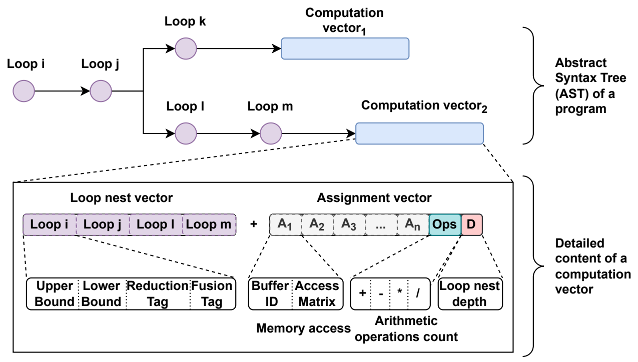
Loop i
Loop j
Loop k
Computation vector$_1$
Loop l
Loop m
Computation vector$_2$
Abstract Syntax Tree (AST) of a program
Loop nest vector
Assignment vector
Loop i | Loop j | Loop l | Loop m
A_1 A_2 A_3 ... A_n Ops D
Upper Bound | Lower Bound | Reduction Tag | Fusion Tag
Buffer ID | Access Matrix
+ - * /
Loop nest depth
Memory access
Arithmetic operations count
Detailed content of a computation vector

Figure 4. Program characterization structure.

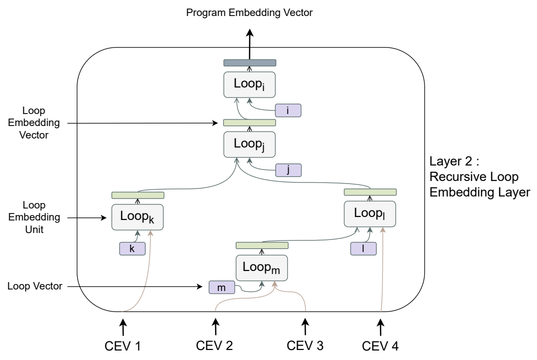

Figure 6. The architecture of the model's second layer.

5.1 Computation Embedding Layer

The first layer of the model extracts an embedding from the *computation vectors*. Each computation vector is processed by a feedforward neural network, as shown in Figure 5, in order to be transformed into a fixed size *computation embedding vector* containing abstract attributes, specific to each computation. We call this vector a CEV (Computation Embedding Vector).

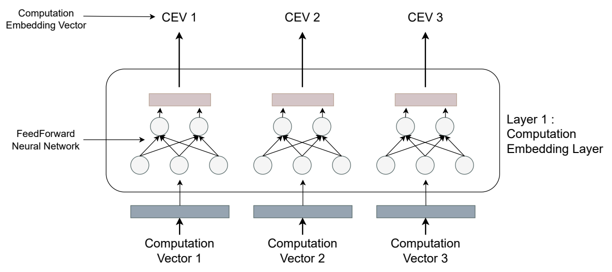
Computation Embedding Vector
CEV 1
CEV 2
CEV 3
FeedForward Neural Network
Layer 1 : Computation Embedding Layer
Computation Vector 1
Computation Vector 2
Computation Vector 3

Figure 5. The architecture of the model's first layer.

5.2 Recursive Loop Embedding Layer

This layer is recursive following the hierarchy of the program. For each loop level, and starting with the innermost level, a *loop embedding unit* is created to represent a loop level, as shown in Figure 6.

However, we have made a modification at this level: in addition to passing to each embedding unit the *computation embedding vectors* and the *loop embedding vectors*, we also pass the *loop vector*, which is the vector corresponding to the current loop level in the loop nest representation, and which contains features about the current loop level (loop extents). Figure 7 summarizes the structure of the *loop embedding unit* we use for our model.

$LSTM_A$ represents a long short-term memory (LSTM) network which transforms all the CEVs of the current loop level into a fixed size vector. Similarly, $LSTM_B$ is a separate LSTM network which transforms all the loop embedding vectors of other loop levels nested in the current loop itself (see Loop j in Figure 6 for example). These two LSTM output vectors will be concatenated with the loop vector, and processed by a feedforward neural network to create the current loop embedding vector.

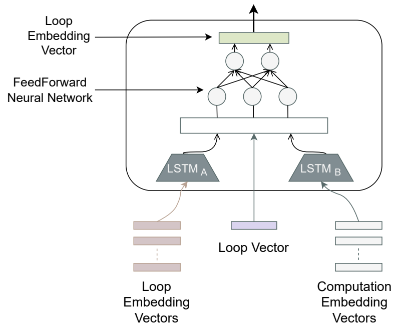

Figure 7. The structure of the *loop embedding unit*.

5.3 Classification Layer

This last layer consists of a feedforward neural network that takes as input the *program embedding vector* produced by the previous layer and predicts the k-best classes of loop interchanges, ordered according to the speedup they provide (a loop interchange class represents a pair of loops to be interchanged). The output of this layer, which represents the value predicted by the model, is represented as a vector containing the duplication of 'k' vectors, each composed of the set of possible loop pairs.

The total number of loop pairs for a program depends on its depth. However, to avoid dealing with a variable output size, we chose to consider 7 loop levels at most. In case the model encounters a program with more loop levels, it predicts the best interchange for its first 7 outermost loops, aiming mainly to optimize code parallelization. We will thus have 21 possible pairs of loops ($C_7^2 = 21$). Furthermore, by adding a class for the case where "no interchange" needs to be applied, each vector in the output vector will have 22 elements.

Figure 8 illustrates the architecture of this last layer taking as an example k=3. According to this example, the model has predicted that the best option is to interchange the first and the second loop (I(L0, L1)). The second best option is not to perform any interchange, thus keeping the current order of the program. The third best option is to interchange the first and the third loop (I(L0, L2)).

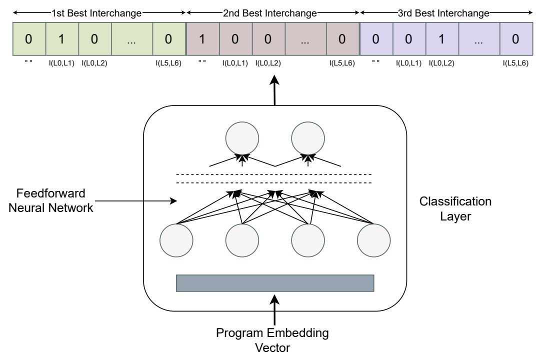

Figure 8. The architecture of the classification layer with k=3.

6 Evaluation

6.1 Experimental Settings

Our model was trained on data generated using the methodology described in Section 3. We randomly extracted 80% of the dataset for the training, 10% for validation and 10% as a test set. Additionally, we check that the distribution of loop interchange instances, and program depths were fairly distributed. That dataset was generated using a multi-core dual-socket machine. Each socket is a 12-core Intel Xeon E5-2695 v2 CPU with 128 GB RAM total. The same machine was used to run all of our experiments. We trained the model on a separate machine (Nvidia DGX A100 machine that has 8 Nvidia A100 GPUs with 80GB of memory for each). We only used a single GPU for the training.

To test the performance of the model, both synthetic and real-world benchmarks have been used. The latter is composed of the same benchmarks used to evaluate the Tiramisu cost model [5]. It contains Tiramisu programs covering different areas: linear algebra (matmul, for matrix multiplication; and jacobi (1D and 2D) and seidel2d, for solving linear systems with the jacobi and the Gauss-Seidel method respectively), image processing (blur for blurring images) and simulation domain (heat2d and heat3d, for heat propagation simulations in 2D and 3D spaces, respectively). Instead of passing only one input to each benchmark, we pass four different inputs. The goal here is to evaluate the model with different input sizes for each benchmark. These inputs are labeled as follows: MINI, SMALL, MEDIUM and LARGE. The full list of benchmarks and their input sizes is presented in Table 4 (Appendex A). The benchmark *seidel2d-MINI* in this table, for example, is *seidel2d* with the input size MINI ($32 \times 49 \times 49$).

We also used synthetic data to evaluated the model. It is composed of 14161 synthetically generated programs. These programs are generated randomly using the same methodology used to generate the dataset but they were not used during the training. The goal here is to provide a wider range of program patterns to further test our model.

We used the Cross-Entropy Loss function. This loss function computes the difference between the predictions made by our model and the ground truth values provided in the dataset, according smaller values to more accurate predictions. Both the dropout method and the weight decay have been used to ensure the regularization of the model, and avoid over-fitting.

To evaluate the model, the accuracy classification score has been used. By definition, it counts the number of samples where the prediction ($\hat{y}$) is identical to the target values (y), or ground truth. Each time the prediction is performed correctly ($\hat{y} = y$), the indicator function (a.k.a. characteristic function), noted below as $\mathbb{1}$, returns 1. If not, it will return 0. Adding up the resulting values of these function over our dataset, and dividing them by the total number of the samples (n), measures how correct the predictions were over the dataset, which we call the accuracy of the prediction. The following equation illustrates the formula used to compute the accuracy:

$$accuracy(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}(\hat{y} = y)$$

As for our model, the output vector contains k sub-vectors ($\hat{y}_1, \hat{y}_2 ... \hat{y}_k$) presenting k ordered predictions of the loop interchanges (in our experiments, k is equal to 5). The ground truth (y) here represents the best loop interchange, which is found by compiling and running the program being optimized. Basing on these values, many variations of accuracy are used for more representative evaluation.

- **1-shot accuracy:** it compares the first sub-vector ($\hat{y}_1$) of the output of our model with the best loop interchange (y). Here, we measure the ability of our method to correctly predict the best instance, directly, in a single shot.

$$accuracy_1(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}(\hat{y_1} = y)$$

- **2-shots accuracy:** Here, we define a slightly larger margin of error. The model's prediction is considered accurate, or correct, if the best loop interchange is predicted in either the first ($\hat{y}_1$) or the second ($\hat{y}_2$) sub-vector of the output. This accuracy should be bigger than the first one, as more data points, that were not predicted correctly at first, are now considered correct.

$$accuracy_2(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}(\hat{y_1} = y \vee \hat{y_2} = y)$$

- **Any-shot accuracy:** using this accuracy, we want to measure whether the model is able predict the best loop interchange at all. This is useful during the first stages of the training of the model, since it proves that the model is capable of predicting the best loop interchange, even if it takes many shots to get it right.

$$accuracy_k(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} \mathbb{1}(\exists i \in 1..k : \hat{y_i} = y)$$

6.2 Tests

6.2.1 Evaluating the Model's Accuracy. In this section, we evaluate the accuracy of our loop interchange model on the benchmarks and compare it with the accuracy of the deep learning cost model currently used by the Tiramisu auto-scheduler. We also compare our model to Pluto [8], a state-of-the-art polyhedral compiler that does not use machine learning for its cost model. We report the 1-shot and 2-shots accuracy for our proposed model and for the current Tiramisu model. Since Pluto only predicts one loop order for each program, we only report the 1-shot accuracy for Pluto.

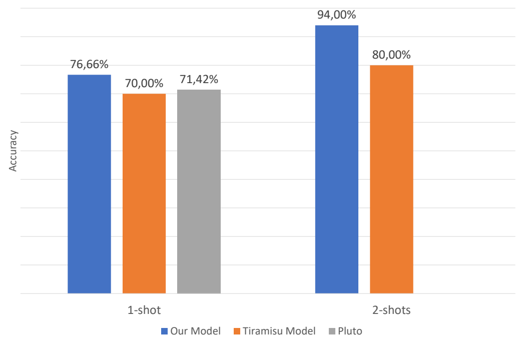

Figure 9. The accuracy of the loop interchange model vs Tiramisu's model vs Pluto, on the benchmarks.

As shown on Figure 9, our model was able to achieve higher accuracy than the Tiramisu model (+6.66% better with 1-shot accuracy, and +14% with 2-shots accuracy). It also outperformed Pluto by 5.24% (1-shot).

The difference with Pluto is because Pluto uses a linear cost function (objective function for its ILP solver), and therefore a non-linear data-driven cost function, such as the one we propose, would capture the complexity of the optimization space much better. Moreover, Pluto doesn't consider loop sizes in its cost function which leads, in some cases, to sub-optimal solutions. For example, Pluto predicts for the different implementations of jacobi-2d that no loop interchange should be applied, while our model detects, for the mini, small, and medium sizes, that moving a loop with bigger extent to the outermost loop level is more effective and thus, applying this loop interchange gives a speedup of 1.6.

While our model was trained on a smaller amount of data compared to Tiramisu's model (208919 data points compared to 1.8M data points for Tiramisu's model), our model has the advantage of being specialized. i.e., trained to only predict the best loop interchanges. This is in contrast to the Tiramisu model which is trained to be general to all code transformations (it predicts the expected speedup of any combination of code and code transformations).

6.2.2 Evaluating the Loop Interchange Exploration Time. In this section, we compare the execution time spent by our model, and Tiramisu's cost model [5] when exploring loop interchange in Tiramisu's auto-scheduler. We use both models to predict the loop interchanges in Tiramisu's auto-scheduler. The evaluation is performed on the benchmarks.

The Tiramisu auto-scheduler uses a tree-structured search space to model the problem of finding the best code transformations and their parameters. In each level of the tree, a new code transformation is explored. A deep-learning based cost model is used as a cost model. This cost model estimates the speedup of each schedule. The search algorithm, beam search, picks the best N nodes having the best predicted speedups, and explores them further in the next level. The optimizations explored by the auto-scheduler are: loop fusion, interchange, skewing, 2D tiling, 3D tiling, unrolling and parallelization.

In the experiments of this section, we measure the time spent in exploring loop interchange when expanding the search tree (i.e., the time spent to explore all the possible loop interchanges and pick the best ones). The results are reported in Figure 10. It shows the speedups in exploring loop interchange when our model is used compared to using the cost model of tiramisu (values higher than 1 mean that exploration using our model is faster).

The figure shows that, on all the benchmarks, our model helps the autoscheduler find the best loop interchange faster (4× median speedup). In the case of matmul for example, finding the best loop interchange using our model is 4× faster

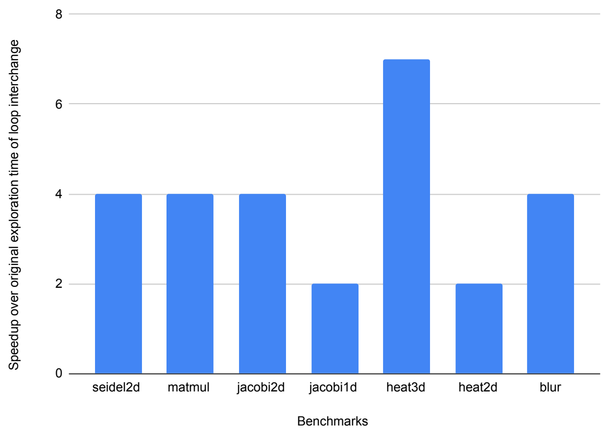

Figure 10. Speedup in finding the best loop interchanges in the benchmarks

than finding the best loop interchange using Tiramisu's cost model. This is because our model needs to be executed only once, while Tiramisu's cost model needs to be executed 4 times (one time to predict the speedup when no loop interchange is applied, and three other times to predict the speedup for each one of the possible loop interchanges in matrix multiplication which has three loop levels).

While our model needs to be executed only once, regardless of the number of loop levels of the program being optimized, Tiramisu's cost model needs to be executed once for each possible loop interchange in the program, with the number of possible loop interchanges growing quadratically with the number of loop levels.

6.2.3 Evaluating the Speedups of Generated Code. In this section, we evaluate the speedups of code generated when our loop interchange model is used.

To do this, we integrate our model into Tiramisu's auto-scheduler. We use it to predict the best loop interchanges to be explored in the next level when expanding the search tree. As for the other transformations, we use a perfect cost model, which returns the ground-truth, to pick the best transformations (the ground-truth in this case is the speedup obtained by compiling and executing the optimized program).

We compare the speedups found using this setup with the speedups found using the same auto-scheduler but when a perfect cost model is used for all transformations. The goal here is to measure whether the loop interchange model can guide the auto-scheduler to find the best schedules.

Using a beam size equal to three, we recorded the final schedule (sequence of chosen transformations) and speedup of the generated code using the two autoschedulers described above (the one with our model, and the one with the perfect model). Table 2 reports these results. The third and forth columns report the code speedups obtained when our model is used to perform the auto-scheduling. In column three, the baseline is the original, unoptimized code. Values higher than 1 mean that the auto-scheduler could accelerate the original code. In column four, the baseline is the schedule found using a perfect model. Values equal to 1 mean that the auto-scheduler with our model found the same speedup as the auto-scheduler with a perfect cost model.

The second column provides, for each benchmark, the schedule found when our model was used. The schedule is expressed as a list of transformations, where each transformation is applied on certain loops and might have parameters. The transformations are Skewing (S), Interchange (I), 2D Tiling (T2), 3D Tiling (T3), Parallelization (P), and Unrolling (U). The loops are labeled as L0, L1, L2, etc. where L0 indicates the outermost loop. In jacobi2d-LARGE for example, the schedule found is the combination of a skewing applied on the two outermost loops (L0 and L1) with the skewing parameters (2,1), and tiling of the two outermost loops with the tiling parameters (32,32). The loop level L1 in the final optimized code is parallelized along with the unrolling of the loop level L4.

The third column of the table shows that the speedups obtained over the original unoptimized code range from 1.0 to 467.94% (1.0 in this case means that the program was neither optimized, nor degraded). Loop interchange was applied in 9 out of 28 benchmarks. In matrix multiplication for example, the loop interchange helped improving the memory access patterns and data locality. In the case of blur-LARGE, loop interchange enabled better parallelism, since the outermost loop in the original code has a small extent and therefore applying an interchange would help in obtaining a better parallelism as detailed in the introduction. Furthermore, our model could detect that applying this interchange is not necessary for the medium size input, and therefore decided to only parallelize the outermost loop to optimize code. The model has also successfully predicted that loop interchange is not necessary in many benchmarks. This is important since many programs do not need any loop interchange.

In all of the benchmarks (100%), our model led the auto-scheduler to find the same schedules found by the auto-scheduler with a perfect cost model. As a result, the speedups obtained with our model and those obtained when the perfect model were identical. Note that the schedules found by the perfect model are not the optimal schedules. This is because the search space exploration method used is beam-search with a beam size equal to three. While this allows finding good schedules, it does not search the whole space, and favors schedules that provide an immediate benefit over those that provide a delayed benefit. Note also that in many cases, the autoscheduler with a perfect cost model finds many equivalent schedules that all lead to similar performance. An arbitrary one of these is chosen. With these notes said, this experiment shows that while our model is not perfect, it is

Table 2. Results (schedules and speedups) of the auto-scheduling of the benchmarks using our loop interchange model.

Benchmark	Schedule found with our model	Speedup over unoptimized code	Speedup over perfect autoscheduler
seidel2d-MINI	S(L1,L2,3,1) (S(L0,L1,3,1)U(L2,8))	1.47	1
seidel2d-SMALL	S(L1,L2,2,1)T2(L1,L2,32,32)	1.43	1
seidel2d-MEDIUM	S(L1,L2,3,2)T2(L1,L2,32,64)	1.56	1
seidel2d-LARGE	S(L0,L1,2,1)P(L1)U(L2,4)	14.89	1
matmul-MINI	I(L1,L2)U(L2,16)	4.91	1
matmul-SMALL	I(L1,L2)P(L0)T2(L0,L1,32,32)U(L4,16)	10.28	1
matmul-MEDIUM	I(L1,L2)P(L0)T2(L1,L2,64,128)	86.62	1
matmul-LARGE	I(L1,L2)P(L0)T2(L1,L2,32,128)	**467.94**	1
jacobi1d-MINI	U(L1,8)	1.28	1
jacobi1d-SMALL	S(L0,L1,2,1)	2.04	1
jacobi1d-MEDIUM	S(L0,L1,2,1)	2.59	1
jacobi1d-LARGE	S(L0,L1,3,1)	2.1	1
jacobi2d-MINI	I(L1,L2)S(L0,L1,3,2)U(L2,16)	3.09	1
jacobi2d-SMALL	I(L1,L2)S(L1,L2,1,1)	3.12	1
jacobi2d-MEDIUM	S(L1,L2,1,1)	1.91	1
jacobi2d-LARGE	S(L0,L1,2,1)P(L1)T2(L0,L1,32,32)U(L4,8)	13.82	1
heat2d-MINI	I(L0,L1)U(L1,8)	1.34	1
heat2d-SMALL		**1.0**	1
heat2d-MEDIUM	P(L0)T2(L0,L1,64,128)	1.79	1
heat2d-LARGE	P(L0)T2(L0,L1,32,64)	8.09	1
heat3d-MINI	S(L0,L1,3,1)U(L3,8)	1.82	1
heat3d-SMALL	I(L1,L2)U(L3,16)	1.35	1
heat3d-MEDIUM	S(L2,L3,1,2)	1.25	1
heat3d-LARGE	S(L0,L1,2,1)P(L1)U(L3,8)	18.6	1
blur-MINI	U(L2,8)	2.39	1
blur-SMALL		1.0	1
blur-MEDIUM	P(L0)	1.92	1
blur-LARGE	I(L0,L1)P(L0)U(L2,8)	6.8	1

enough to enable the autoscheduler to find the same quality of schedules that it would have found using a perfect model.

6.3 Other Design Choices

6.3.1 Single Prediction Model (1-best). During the first stages of our work, we tried to predict directly the best loop interchange (instead of predicting the best k loop interchanges as we do in out final design). The classification accuracy scores obtained using this model are presented in the following table 3.

Table 3. Accuracies of the single prediction model

Test set	Benchmark
80.163%	83.33%

Comparing these results with the 1-shot accuracies of the k-best model, we can see that numbers are close. The 1-shot accuracy of the k-best (with k = 5) model is 0.2% better than the single prediction model in the test set, whereas it is 3.33% worse in the benchmarks (this difference is due to one wrong prediction in the benchmarks, since the number of benchmarks is small). Overall, the results are close with a slight advantage for the 1-best model.

However, comparing its results with the 2-shots accuracy, we see a larger difference. On both benchmark and test set, the 2-shots accuracy of the k-best model (with k = 5) is better. Our proposed model reaches a 2-shots accuracy of 94.64% on the test set, and 86.66% on the benchmark. Our proposed model is 14% better than the single prediction model in the test set and 3.33% in the benchmarks.

The accuracy of the k-best model with 2-shots surpass the accuracy of single prediction model. Even if the performance of the latter are slightly better for 1-shot, the use of the k-best model is better when combined with a more general search algorithm. First, because the k-best model has higher accuracies and second because when used in an auto-scheduler, it proposes multiple loop interchanges to be explored further. This is useful because while certain loop interchanges do not provide the best speedup when evaluated early in the exploration, they provide the best speedups when combined with other optimizations later on during the exploration. The code presented in the introduction is an example of this, where applying loop interchange alone reduces data locality and decreases performance but when this loop interchange is combined with parallelization the combination provides the best speedup.

Having a model predicting only one loop interchange (single prediction model) forces the search function to pick one schedule in the loop interchange level. The space explored later is significantly reduced, as only schedules using this loop interchange are considered.

6.3.2 Multi-Label Classification. We previously presented our model's output format, which consists in assigning the input program to one of the 22 possible classes (these 22 classes form a vector). We duplicate this vector 'k' times to have the k-best loop interchanges. In addition to this format, we also explored the multi-label classification format. In this type of classification, an object can belong to several classes at the same time. In our case, the goal is to predict the ability of a loop to be interchanged. Thus, each loop level is represented as a class, and since the model aims to predict the two loops to be interchanged, it assigns the program to two classes. The two classes (two loops) that have the highest values in the output vector represent the two loops to be interchanged. As we consider 7 loop levels at most, the output is a vector of 7 elements. However, to consider the case where no interchange should be applied, we chose to use a *threshold* as a limit value to define whether any loops should be interchanged at all. If the highest predicted value is lower than this *threshold*, we consider that no loop should be interchanged, and thus predict that no loop interchange should be performed, by setting all elements of the vector to zero.

Figure 11 shows a comparison of the accuracy of two models: the single shot model (output vector of 22 elements) and the multi-label one (output vector of 7 elements). In this figure, we show the accuracy of the multi-label model for different values of the threshold ranging from 0 to 1. To simplify the comparison, we also represent in the same figure the accuracy of our proposed loop interchange model (which is constant because this model does not require the definition of a *threshold*). The models are evaluated on the test set, as well as on the benchmarks.

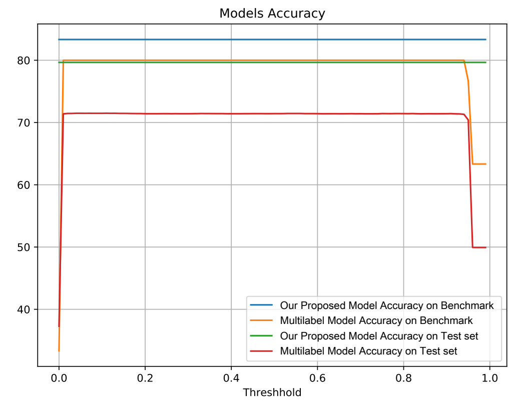

Figure 11. Our proposed model and multi-label classification model accuracy.

For the multi-label classification model, a threshold value close to 0 implies poor performance, because it enforces the model to consider that an interchange must be applied, and that the case of "no loop interchange" cannot occur. However, our test set contains approximately 50% of programs where no interchange should be applied. On the opposite, a value close to 1 forces the model to only consider the case where no loop interchange is applied. Thus, according to the graph, the model has 50% accuracy for these values. This accuracy is due to the high number of programs where applying no interchange is the best decision. Furthermore, no value of the threshold allows the multi-label classification model to reach the performance obtained by our proposed model.

6.4 Summary of Evaluation Results

Our evaluation has shown that the proposed k-best model has a higher accuracy than the current Tiramisu cost model. It also has shown that finding the best loop to interchange using our proposed model is less time consuming compared to exploring the space using the current Tiramisu's cost model. With this increase in the exploration speed, the proposed model did not lead to a loss in the speedup of generated code. The speedup of code generated when our model is used is comparable to the speedup obtained when Tiramisu's current model is used.

We have also explored different design choices to implement the loop interchange model. The single prediction model served as a starting point, modeling the loop interchange prediction problem as a classification problem. Then, we explored the multi-label classification format. The latter, having not shown better results, was discarded. However, in order to integrate our model to the Tiramisu auto-scheduler which explores the search space using a heuristic, we modified the single prediction model so that it can predict the 5 best loop interchanges to apply on a program (k-best model).

The k-best model turned out to be much better. It was able to increase the 2-Shots accuracy of loop interchange prediction by up to 14%.

7 Related Work

Previous research has tried to use analytical models to automatically apply a loop interchange when profitable. Allen & kennedy [3], try to enable the application of vectorization by pushing the dependence-free loops to the innermost position. In GCC [1], two loops are interchanged if this transformation allows better data locality, creates more invariant memory references, or makes all memory accesses contiguous. However, this process is done for each pair of loops. This means that the compilation time increases proportionally with the number of loop levels in every loop nest in the program.

Many polyhedral compilers such as Pluto [8], LLVM Poly [14], and Tensor Comprehensions [18], use the Pluto algorithm [7], which uses integer linear programming (ILP) to find the best order of execution of statements and thus automatically apply loop interchange. The main idea of this algorithm is to minimize the distance between producer and consumer statements, which pushes parallel loops to the outermost levels. With this approach, Pluto always prioritizes maximizing data locality and outermost parallelism when this is legal. However, this can lead to a decrease in performance if the best optimizations depend on other factors (such as the loop extent or the data layout being used).

Other compilers have used machine learning to automatically optimize code. Those compilers include Tiramisu [5] which uses a deep learning based cost model, jointly with a tree search algorithm, to explore the search space and find the best set of code transformations to apply. AutoTVM [10] uses a deep learning model that predicts the runtime of tensor programs and select code transformations parameters. Similarly, Halide [2] uses a feedforward neural network to predict the execution time of programs and explore the space of code transformations. Our work is complementary to these auto-schedulers. Our model can be integrated in the search techniques used by these models to reduce the overall search space exploration time.

Previous research has also addressed the problem of selecting transformation parameters using machine learning. Wang and O'Boyle [19] used a machine learning model to determine the best number of threads for an already parallelized program. DeepTune [11] proposed a neural network to predict the mapping of OpenCL kernels. In the same context, Rahman et al. [17] proposed an artificial neural network in order to select the best tile sizes. Li and Garzaran [16] proposed a classification system which determines the best loop levels to tile and the best tile sizes. Our proposed model is specialized in selecting the best loops to interchange.

8 Conclusion

We presented a deep learning model designed to predict the best loops to interchange in a program. By modeling this problem as a classification problem, our model takes as input simple and high-level attributes of a program and predicts, in order, the k-best loop interchanges to apply. We evaluated the proposed model on a benchmark of Tiramisu programs and showed an accuracy of 76.66% for 1-shot and 94% for 2-shots. Experiments show that our model outperforms the cost model currently used by the Tiramisu compiler by 6.66% in terms of 1-shot accuracy, and 14% with 2-shots accuracy, while at the same time reducing the total execution time needed for predicting the best pair of loops to interchange. This is without any loss in the speedup of the generated code.

Adapting the proposed model to predict where to apply loop interchange, as well as generalizing it to other architectures and evaluation it outside of the Tiramisu framework are to be explored for future work.

References

[1] 2021. GCC, the GNU Compiler Collection - GNU Project. https://gcc.gnu.org/

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics* 38, 4 (July 2019), 1–12. https://doi.org/10.1145/3306346.3322967

[3] Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach* (1st edition ed.). Morgan Kaufmann, San Francisco.

[4] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven Van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 138–149. https://doi.org/10.1109/PACT.2015.17 ISSN: 1089-795X.

[5] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. *arXiv:2104.04955 [cs]* (April 2021). http://arxiv.org/abs/2104.04955 arXiv: 2104.04955.

[6] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2018. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *arXiv:1804.10694 [cs]* (Dec. 2018). http://arxiv.org/abs/1804.10694 arXiv: 1804.10694.

[7] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J Ramanujam, Atanas Rountev, and P Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. *International Conference on Compiler Construction (ETAPS CC)* (April 2008). https://www.csa.iisc.ac.in/~udayb/publications/uday-cc08.pdf

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on*

Programming Language Design and Implementation (PLDI '08). Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *arXiv:1802.04799 [cs]* (Oct. 2018). http://arxiv.org/abs/1802.04799 arXiv: 1802.04799.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. Learning to Optimize Tensor Programs. *arXiv:1805.08166 [cs, stat]* (Jan. 2019). http://arxiv.org/abs/1805.08166 arXiv: 1805.08166.

[11] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Portland, OR, 219–232. https://doi.org/10.1109/PACT.2017.24

[12] P. Feautrier. 1988. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing (ICS '88)*. Association for Computing Machinery, New York, NY, USA, 429–441. https://doi.org/10.1145/55364.55406

[13] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[14] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (Dec. 2012), 1250010. https://doi.org/10.1142/S0129626412500107 Publisher: World Scientific Publishing Co..

[15] Ken Kennedy and Kathryn S. McKinley. 1992. Optimizing for parallelism and data locality. In *Proceedings of the 6th international conference on Supercomputing (ICS '92)*. Association for Computing Machinery, New York, NY, USA, 323–334. https://doi.org/10.1145/143369.143427

[16] Xiaoming Li and María Jesús Garzarán. 2006. Optimizing Matrix Multiplication with a Classifier Learning System. In *Languages and Compilers for Parallel Computing (Lecture Notes in Computer Science)*, Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan (Eds.). Springer, Berlin, Heidelberg, 121–135. https://doi.org/10.1007/978-3-540-69330-7_9

[17] Mohammed Rahman, Louis-Noel Pouchet, and Ponnuswamy Sadayappan. 2010. Neural Network Assisted Tile Size Selection. (Nov. 2010), 15.

[18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv:1802.04730 [cs]* (June 2018). http://arxiv.org/abs/1802.04730 arXiv: 1802.04730.

[19] Zheng Wang and Michael F P O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. (April 2009), 10.

A Benchmark Sizes

Table 4 details the benchmarks used in Section 6, and their input sizes.

Received 2022-11-10; accepted 2022-12-19

Table 4. Input data sizes for the benchmarks

Benchmark	Input Sizes
seidel2d-MINI	32 x 49 x 49
seidel2d-SMALL	48 x 129 x 129
seidel2d-MEDIUM	128 x 449 x 449
seidel2d-LARGE	512 x 2049 x 2049
matmul-MINI	16 x 32 x 48
matmul-SMALL	64 x 96 x 128
matmul-MEDIUM	192 x 256 x 320
matmul-LARGE	1024 x 1280 x 1536
jacobi1d-MINI	64 x 33
jacobi1d-SMALL	96 x 129
jacobi1d-MEDIUM	256 x 449
jacobi1d-LARGE	1024 x 2049
jacobi2d-MINI	64 x 49 x 49
jacobi2d-SMALL	48 x 129 x 129
jacobi2d-MEDIUM	256 x 385 x 385
jacobi2d-LARGE	1024 x 1537 x 1537
heat2d-MINI	17 x 65
heat2d-SMALL	129 x 65
heat2d-MEDIUM	257 x 257
heat2d-LARGE	1025x1537
heat3d-MINI	1024 x 129 x 129 x 129
heat3d-SMALL	256 x 49 x 49 x 49
heat3d-MEDIUM	48 x 17 x 17 x 17
heat3d-LARGE	96 x 33 x 33 x 33
blur-MINI	4 x 17 x 33
blur-SMALL	4 x 33 x 97
blur-MEDIUM	4 x 97 x 257
blur-LARGE	4 x 513 x 1025