A Deep Learning Based Cost Model for Automatic Code Optimization

Riyadh Baghdadi [1 2] **Massinissa Merouani** [3] **Mohamed-Hicham Leghettas** [3] **Kamel Abdous** [3] **Taha Arbaoui** [4] **Karima Benatchba** [3] **Saman Amarasinghe** [1]

Abstract

Enabling compilers to automatically optimize code has been a longstanding goal for the compiler community. Efficiently solving this problem requires using precise cost models. These models predict whether applying a sequence of code transformations reduces the execution time of the program. Building an analytical cost model to do so is hard in modern x86 architectures due to the complexity of the microarchitecture. In this paper, we present a novel deep learning based cost model for automatic code optimization. This model was integrated in a search method and implemented in the TIRAMISU compiler to select the best code transformations. The input of the proposed model is a set of simple features representing the unoptimized code and a sequence of code transformations. The model predicts the speedup expected when the code transformations are applied. Unlike previous models, the proposed one works on full programs and does not rely on any heavy feature engineering. The proposed model has only 16% of mean absolute percentage error in predicting speedups on full programs. The proposed model enables TIRAMISU to automatically find code transformations that match or are better than state-of-the-art compilers without requiring the same level of heavy feature engineering required by those compilers.

1 Introduction

Writing high-performance software is essential in many areas from machine learning to science and engineering. In nuclear physics, for example, researchers need to perform large scale simulations to study the properties of matter. A highly optimized implementation of these simulations can be orders of magnitude faster compared to an unoptimized implementation. In deep learning, an optimized implementation of a state-of-the-art neural network such as XLNet (Yang et al., 2019) is $1.8\times$ faster than the equivalent PyTorch implementation. Writing such a highly optimized code requires ninja programmers and is time-consuming while the results are error-prone, less understandable, and non-portable. One of the longstanding goals in the compiler community is to develop compilers that can automatically optimize high-level code. These compilers automatically apply code transformations to make the code run faster; thus, avoiding the need for manual low-level program tuning. They provide greater productivity, portability, and high performance, and will be directly accessible by domain scientists.

[1]Massachusetts Institute of Technology [2]New York University Abu Dhabi [3]Ecole Nationale Superieure d'Informatique [4]University of Technology of Troyes. Correspondence to: Riyadh Baghdadi <baghdadi@nyu.edu>.

Automatically generating efficient code for high-performance systems is a tedious task. In order for the compiler to generate efficient code, two problems have to be solved. First, a large set of code transformations and a mechanism to apply them to programs need to be provided. Examples of such transformations include loop fission, fusion, parallelization, and vectorization. Second, the right sequence of code transformations from this large set has to be chosen. The selected code transformations must preserve the program semantics and provide the highest performance for the input program. While state-of-the-art-compilers have shown success in solving the first problem (i.e., the ability to provide a large set of transformations and correctly apply a selected sequence of transformations (Wolf & Lam, 1991; Bondhugula et al., 2008; Trifunovic et al., 2010; Grosser et al., 2014; Lefebvre & Feautrier, 1998; Quilleré & Rajopadhye, 2000)), they still do not successfully solve the second problem (i.e., selecting the sequence of transformations that will provide the best performance).

The problem of selecting the right sequence of code transformations can be modeled as a search problem that can be solved in three steps. In the first step, the compiler uses a search technique to explore the space of possible code transformations. The result of this step is a set of candidates where each one is a sequence of code transformations. In the second step, the compiler checks the validity of each candidate (i.e., checks that applying the transformations does not change the program semantics). In the third step, the compiler evaluates the valid candidates and chooses the one that minimizes the execution time. This evaluation can be done by running each candidate on the target hardware to obtain the exact speedup. However, this is not a feasible solution in practice as running a program takes a considerable amount of time. Moreover, the target hardware may not be available at compile time. Another way to evaluate a candidate is by using a cost model to predict the speedup.

Designing cost models manually is known to be a hard task (Trifunovic et al., 2009; Bachir et al., 2013). This is mainly due to the diversity of hardware architectures and their complexity (out-of-order execution, complex memory hierarchies, data prefetching, etc.). Complex interactions between code transformations make the problem more complicated. Recently, cost models, such as Ithemal (Mendis et al., 2018) and Halide (Adams et al., 2019), have demonstrated how to overcome some of this complexity by using deep learning. While these state-of-the-art cost models are more accurate, they are limited in two ways: Ithemal (Mendis et al., 2018) only predicts throughput for basic blocks of assembly code (instead of full programs). It also assumes that data is always in cache. The cost model in Halide (Adams et al., 2019) requires heavy feature engineering (it uses 54 complex program features). Designing such features is tedious, error-prone, and time-consuming.

In this paper, we propose a novel DNN-based cost model that avoids the problems of previous work. Our model operates on full programs expressed in a high-level language (not just basic blocks). It takes into consideration not only memory accesses to the cache but also to the main memory. Moreover, it does not require heavy feature engineering. The proposed cost model takes the original unoptimized code and a sequence of code transformations and predicts the speedup that these transformations would yield when applied. The model is designed for CPUs and is integrated in the Tiramisu compiler (Baghdadi et al., 2019), a compiler for the Tiramisu domain-specific language (DSL). Because this model is a regression model, it allows the compiler to select the best transformation candidates by ranking the candidates selected by a search technique.

Contributions In summary, the contributions of this paper are:

- A novel deep-learning-based cost model for code optimization. This cost model is a *regression* cost model, operates on *full programs*, and *does not rely on extracting complex features*.
- An implementation of the proposed model and an integration into a search approach to enable the Tiramisu compiler to automatically search for the best code transformations.
- We evaluate the proposed model and show that it has a low error rate reaching 16% mean absolute percentage error. We show also that it enables Tiramisu to automatically find code transformations that match or outperform state-of-the-art compilers.

2 Tiramisu Embedded DSL

Tiramisu (Baghdadi et al., 2019) 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. The first part of a Tiramisu program specifies the algorithm without specifying how it should be optimized. The second part specifies which code transformations to apply and how the results of computations should be stored. Tiramisu uses a mathematical model known as the polyhedral model internally (Feautrier, 1988; Baghdadi et al., 2015a; Bondhugula et al., 2008; Baghdadi et al., 2015b; 2019) to represent code, code transformations, and to reason about the correctness of code transformations. The following code shows an example of a convolution algorithm written in Tiramisu.

```
1 // Declare the iterators.
2 var n(0, batch), fout(0, out_features), fin
      (0, in_features), y(0, H-2), x(0, W-2),
       k0(0, 3), k1(0, 3);
3 // Algorithm.
4 conv(n, fout, y, x) += weights(fout, fin, y
      , x) * input(n, fin, y + k0, x + k1);
```

The iterators in line 2 define the loop bounds around the `conv` computation. The algorithm is semantically equivalent to the following code.

```
for (n in 0..batch)
 for (fout in 0..out_features)
  for (y in 0..H-2)
   for (x in 0..W-2)
    for (fin in 0..in_features)
     for (k0 in 0..3)
      for (k1 in 0..3)
       conv[n, fout, y, x] += weigths[fout,
     fin, y, x] * input[n, fin, y+k0, x+k1];
```

The next code shows an example of code transformation commands that can be applied to the previous convolution kernel. These commands apply parallelization, loop interchange, tiling, vectorization, and unrolling.

```
// Provide the code transformation commands.

conv.parallelize(n);
conv.interchange(fout, fin);
conv.tile(y, x, 32, 32);
conv.vectorize(fout, 8);
conv.unroll(k0); conv.unroll(k1);
```

Currently, in Tiramisu, a developer has to provide the previous sequence of code transformations manually. Our goal is to automate finding that sequence. We do this by developing a cost model that predicts the speedup of using a given transformation or any sequence of valid transformations. For example, the model can be used to predict whether combining parallelization, loop interchange, and loop tiling is useful. In addition, the model can be used to choose the right arguments for each one of the previous code transformations (e.g., choose the tile sizes).

3 Data Generation

As training DNNs requires a large data set and only a small number of programs have ever been written in Tiramisu, we decided to automatically generate a data set and use it to train the model. We developed a code generator that generates random programs and sequences of code transformations. Each one of these randomly generated programs and code transformations is compiled, executed, and finally, the actual speedup is measured. The speedup is the ratio between the execution time of the original unoptimized program and the optimized one. Each data point in the data set is a triplet of the form (program, a sequence of code transformations, measured speedup).

Random Code Generation A Tiramisu program is a sequence of computations where each computation is an assignment. There are three common patterns of assignments that appear in Tiramisu programs: (1) simple assignments where the right-hand side is a function of input arrays or array values computed previously; (2) stencils; (3) reductions. The random code generator generates sequences of computations where each computation is a variant (or a combination) of the previous patterns. Randomly generated programs are correct by construction. A computation consumes either constants, input arrays, or values computed by previous computations. Code transformations are also generated randomly but specific rules are used to guarantee that code transformations are valid (for example, tiling is not applied if the loop extent is smaller than the tile size).

The random code generator is designed to generate programs that are representative of real programs. The three patterns that the random code generator generates, when combined together, cover all the patterns that we are interested in supporting. The input data is also generated automatically and the size of the input data is chosen randomly. Since the generated programs are not data dependent (no data dependent conditional or data dependent array access), the actual data values are not important. Only the size of the data is important for training the model.

For a given randomly generated program, the random code generator chooses random data sizes, and then generates 32 random sequences of code transformations.

Dataset Construction The total generated dataset has approximately 1.8 million programs. To construct this dataset, we generated 56250 random algorithms. For each algorithm, we generated 32 random sequences of code transformations. Therefore we obtained 56250×32 programs in total. We followed the gold-standard in performance engineering and executed each resulting program 30 times, and retained the median value of the execution times in order to reduce the impact of minor variance in execution times. Since data generation is time consuming, we used a cluster of 16 nodes of multicore CPUs to accelerate data generation. Generating the whole data set took 3 weeks.

4 Program Characterization and Model Architectures

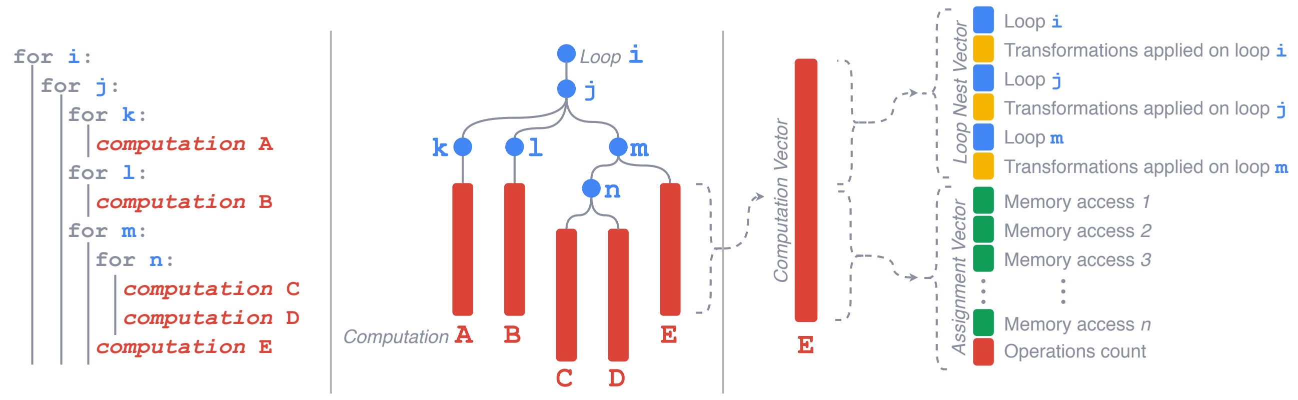
Our cost model is designed to support programs that can be expressed in Tiramisu. The latter is designed for expressing data parallel algorithms that operate over dense arrays using loop nests and sequences of statements. These algorithms are often found in image processing, deep learning, dense linear algebra, tensor operations, and stencil computations. A formal description of programs supported by Tiramisu can be found in (Baghdadi et al., 2019; 2020). Code transformations supported by the proposed model include loop fusion, loop tiling, loop interchange, and loop unrolling which are all challenging. For simpler transformations such as parallelization and vectorization, we use simple heuristics similar to those used by the Halide autoscheduler (Ragan-Kelley et al., 2012). These heuristics mainly parallelize the outermost loops and vectorize the innermost loops when a set of conditions are met.

4.1 Program Characterization

Designing complex hand-engineered features is tedious, error-prone, and time-consuming. Instead of using complex hand-engineered features, we characterize programs by extracting simple high-level information that is stored in a compact variable-size representation.

Our program characterization is based on the AST (Abstract Syntax Tree) representation of programs. A program is characterized as an ordered tree of *computation vectors* as shown in Figure 1b. A *computation vector* is a vector that includes three pieces of information: (1) *loop nest representation*; (2) *assignments representation*; (3) *loop transformation representation*. In the following paragraphs, we describe each of these components, the key features we aim to encode by each one, and how we combine them in a compact way.

Loop Nest Representation The extent of each loop level around the computation is stored in the computation vector (the extent of a loop is calculated based on its lower and upper bounds). An example is shown in Figure 1c. After each loop level extent, for each loop transformation, we insert a boolean tag that represents whether that transformation is applied to that particular loop level. Each transformation is followed by its parameters (when this applies).



(a) Program pseudocode. (b) Program tree representation. (c) Computation vector.

Figure 1: Our characterization of a typical program.

Assignments Representation We represent both the left-hand side and the right-hand side of the assignment. To represent the left-hand side, we store the dimensions and the size of each dimension of the buffer used on the left-hand side. To represent the right-hand side of the assignment (the assignment expression), we store the following information: (1) the memory access pattern (access matrix described later); (2) the ID of each accessed buffer (a number); (3) the count of each arithmetic operation used on the right-hand side (i.e., the number of times each arithmetic operation is used).

We represent the array accesses using an access matrix that stores the coefficients of each array access. This matrix uses exactly the same format used in the polyhedral model to represent array accesses (Paul & Christian, 2011). It only supports arrays that have affine array accesses (i.e., array accesses that are affine in the loop iterators). Supporting only affine accesses is not a problem since TIRAMISU only supports code that has affine accesses. Supporting code that has data-dependent array accesses or non-affine accesses is not within the scope of this paper.

The access matrix has k rows and $n+1$ columns where k is the number of dimensions of the access buffer and n is the loop depth. Each row in the matrix represents an array dimension. Each array dimension is considered to be a linear combination of the loop iterators. Each loop iterator in the matrix is represented by a column. The coefficient of each loop iterator is stored in the column that corresponds to that loop iterator. The last column in the matrix corresponds to constants.

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & -2 \end{bmatrix}$$

For example, the following memory access $A[i_0,\ i_0 + i_1,\ i_1 - 2]$ is represented using the matrix M. In this case, the first dimension of the buffer access is i_0. It can also be written as $1*i_0+0*i_1+0$. It is represented by the first row in the matrix using 1 0 0 where the first column corresponds to iterator i_0, the second column corresponds to i_1 and the last column corresponds to constants. The second access i_0+i_1 is represented with the second row in the matrix 1 1 0. Each *memory access matrix* is succeeded by the identifier of the buffer.

Loop Transformation Representation Each loop transformation can be characterized by two pieces of information: transformation type and its parameters. Since transformations are applied on loop levels, we attach to the representation of each loop level the transformations that are applied to that level (Figure 1c). The transformations that involve changing the structure of the program (e.g. loop fusion) are directly applied to the *program structure representation* that we will describe next.

Program Structure Representation The program is represented as a tree structure where leaves are the *computation vectors* and internal nodes are the loop levels, as shown in Figure 1b.

4.2 Detailed List of Features Composing the *Computation Vector*

Table 1 details the full list of features that constitute the *Computation Vector* presented in Figure 1c. All features are integer values except *Tag* features which are boolean values. In practice, we set $n = 7$ and $m = 21$ where n is the maximum length of the loop nests in our dataset and m is the maximum number of terms used in an assignment. When needed, a zero-padding is added to the *Loop Nest Vector* and *Assignment Vector*.

4.3 Hardware Characterization

The goal of the paper is not to develop a hardware independent model. The proposed model is specific to a particular CPU. The user can build a model for each CPU they want to target. Building a new model does not require a large effort. It can be done simply by running a script that generates new data and retrains the model.

In other words, we do not include any feature to represent the hardware because the model is specific to one and only one hardware architecture. We also do not extract any

Loop Nest Vector	Loop_1	Upper bound, Lower bound, Reduction tag Fusion tag, Interchange tag, Tilling tag, Tilling factor
	Loop_2	Upper bound, Lower bound, Reduction tag Fusion tag, Interchange tag, Tilling tag, Tilling factor
	⋮	⋮
	Loop_n	Upper bound, Lower bound, Reduction tag Fusion tag, Interchange tag, Tilling tag, Tilling factor, Unroll tag, Unrolling factor.
Assignment Vector	Memory access $_1$	Access matrix, Buffer ID.
	Memory access $_2$	Access matrix, Buffer ID.
	⋮	⋮
	Memory access $_m$	Access matrix, Buffer ID.
	Operations count	Number of Additions, Number of Multiplications, Number of Subtractions, Number of Divisions.

Table 1: A detailed listing of the features that compose the *Computation Vector*.

hardware-specific feature from the code for the same reason. Designing a model that works for multiple hardware architectures is beyond the scope of this paper.

4.4 Model Architecture

We model the problem of speedup estimation as a regression problem: given an algorithm and a set of code transformations, our model predicts the speedup expected when applying the suggested code transformations compared to the base program (i.e. without applying code transformations).

We design our cost model's architecture to support the variable size and recursive nature of our program characterization by combining Recurrent and Recursive Neural Networks. Our model's architecture has three layers as shown in Figure 2a.

Computation Embedding Layer All *computation vectors* of the program are processed through a feedforward network after *log*-transforming non-boolean features. This *log*-transformation is necessary since these features have a large dynamic range and most of them are expected to be multiplied with other features to compute the final speedup.

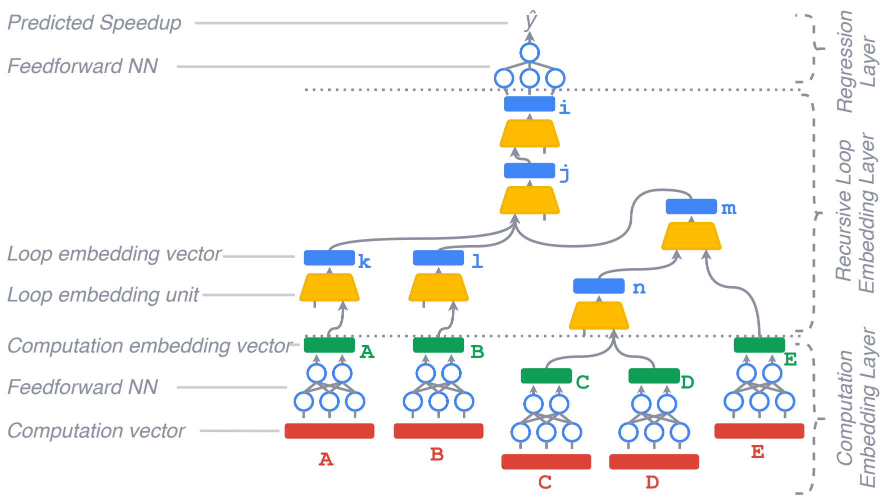
Recursive Loop Embedding Layer The *computation embeddings* resulting from the previous layer are then processed recursively following the tree structure of the program using the *loop embedding unit*. At a given loop level, the loop embedding unit summarizes the program up to that loop into a *loop embedding*. The *loop embedding unit*, as depicted in Figure 2b, is composed of two separate LSTM cells (Hochreiter & Schmidhuber, 1997) and a feedforward layer. At each loop level, the first LSTM is fed with the embedding vectors of computations that are nested directly in that loop level while the second LSTM is fed with the embedding vectors of the previous loop levels that resulted from the previous *loop embedding units*. The two hidden states of the LSTMs are merged using a feedforward layer into a *loop embedding vector*.

The purpose of this recursive embedding is to selectively incorporate information from each computation respecting its positional relations with the other computations. The output of this layer, the *program embedding vector*, is assumed to contain the needed set of automatically extracted features covering the complete program.

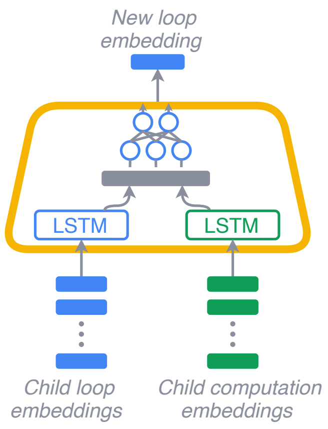
Regression Layer The *program embedding vector* produced by the previous layer is finally fed to a shallow feedforward neural network that performs a regression in order to predict the speedup.

We choose MAPE (Mean Absolute Percentage Error) as an objective function to train our model. The model is implemented in PyTorch (Paszke et al., 2019) and optimized using AdamW (Loshchilov & Hutter, 2017). All the implementation details including layer sizes and training policy can be found in appendix A.1.

Other Neural Network Models Explored We also explored many other alternative architectures for the cost-model, the architecture presented above has the lowest MAPE error on both the test set and benchmarks set. For instance, replacing the *Recursive loop embedding layer* with a simple Recurrent Neural Network that is directly fed with the sequence of *computation embeddings* without taking in consideration the loops hierarchy leads to a relative increase of 1.15x in MAPE of the test set and 1.33x in the benchmarks set compared to the presented architecture. Another straightforward choice of using a simple Feedforwad Neural Network, i.e. totally skipping the *Recursive loop embedding layer* and feeding directly the concatenated *computation embeddings* to the *regression layer*, leads to a relative increase of 1.39x in MAPE of the test set and 1.37x in the bench-



(a) Processing the program presented in Figure 1 through the three layers of the cost-model.



(b) Loop embedding unit.

Figure 2: The cost model architecture

marks set compared to the presented model. In addition, this alternative has the considerable limitation of not supporting variable program sizes and supports only programs that contain up to a certain number of computations (we have set the maximum number of computations to 4 when testing this alternative).

4.5 Level of Feature Extraction

One of the questions that we needed to answer is at which level should the code representation be extracted: directly from the source code or from the transformed code (intermediate representation obtained after applying code transformations)? Our choice was to extract the representation from the TIRAMISU source code for a pragmatic reason. In order for a model that takes transformed code to work, Tiramisu will need to apply the transformations on the program before using the model. Such a step is time-consuming especially because it will be repeated a large number of times (given that the search space is large). Furthermore, transformed code is more complex and therefore is harder to learn from compared to a program and a list of transformations.

5 Search Space Exploration

Finding the best code transformations is a hard combinatorial optimization problem due to the fact that some of the constraints (e.g., interaction between code transformations), and the objective (the speedup in this case), are hard to represent mathematically using the program's features. Thus, the proposed model is used as an objective function estimator to better navigate the search space. However, the used search exploration approach should take into account the estimator's margin of error, thus requiring stochasticity in the search space exploration.

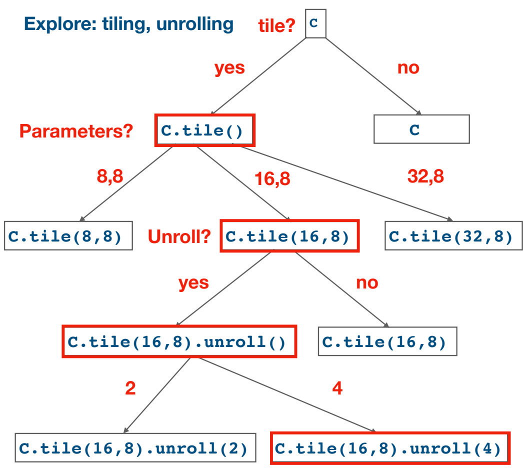


Figure 3: Example of the BS Tree for exploring the tiling and unrolling code transformations

Since the interaction between code transformations is hard to characterize, one of the best ways to model the problem of finding the best code transformations (and their parameters) is to use a tree search. This allows us to use classical tree search algorithms. In this paper, we use Beam Search and MCTS (Monte Carlo Tree Search).

The Beam Search tree (as shown in Figure 3) explores whether to apply a code transformation and which parameters to use for that transformation. At each node of the tree, an evaluation is conducted using the cost model to assess whether the chosen transformations provide a good speedup. In Figure 3, exploring the tree shows that applying tiling with a tile size of (16, 8) and unrolling with a factor of 4 provides the best sequence of code transformations.

MCTS takes advantage of the search tree and takes into account the stochasticity of the model. Particularly, the proposed MCTS combines our model's prediction and an execution of the best evaluated code transformations as a compromise between prediction and execution. MCTS first explores the different branches of the tree and selects a number of *promising code transformations*. To this end, the model is used in each node to obtain an estimate of the speedup. MCTS keeps track of a set of the best evaluated code transformations to execute them (the size of the set is a parameter of the approach). Once the tree is explored, the set of the best code transformations is executed. The advantage of this two-step approach is to accelerate the exploration of the search space using the model and to correct the model's error, if occurred, by executing a limited set of programs and their code transformations. The proposed model is thus used to prune the search space and limit the execution to selected code transformations.

6 Evaluation

To evaluate our cost model: (1) we measure its accuracy on a test set composed of random programs and compare the predicted and the measured speedups on that data set; (2) we measure the speedups obtained when the model is used to search for code transformations in real-world benchmarks; (3) we compare the accuracy of this model with the accuracy of the model used in Halide (Ragan-Kelley et al., 2012), a state-of-the-art model.

The model evaluation and the data collection are performed on 16 identical multi-core CPU nodes. Each node has a dual-socket, each socket is a 12-core Intel Xeon E5-2680v3 CPU, with 128 GB RAM. We used 60% of data for training, 20% for validation, and 20% for testing.

$$MAPE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

Model Accuracy To measure the accuracy of the proposed model, we use MAPE (Mean Absolute Percentage Error), where y and $\hat{y}$ are respectively the measured and the predicted speedups. The MAPE of our cost model on the test set is 16%.

The Pearson correlation coefficient for the proposed model is 0.90, showing that the linear correlation between predicted and measured speedups is strong. In addition, we evaluate the ranking capabilities of the model with the Spearman's rank correlation coefficient, defined as: $r_s(y, \hat{y}) = r\big(rg(y), rg(\hat{y})\big)$ where $rg(y)$ converts the speedups to ranks and r is the Pearson correlation coefficient. The Spearman's rank coefficient of our cost model is 0.95, which shows that the predicted and measured ranks are highly linearly correlated. This property is important when using the model with a search method.

Comparing Predicted and Measured Speedups Figure 4 compares the predicted and measured speedups. To simplify visualization, we use a subset of the test set. This subset is composed of 100 random programs, each with 32 random sequences of code transformations (therefore, the total is 3200 transformed programs). The horizontal axis is the list of 3200 programs. These programs are sorted based on their speedups in ascending order to simplify visualization. As the figure shows, the predicted speedups are close to the measured ones. The error in prediction is lower around the speedup 1 and is higher as the speedup gets further from 1. We will comment more on this behavior later in the section.

Figure 5 investigates the distribution of the model error rates over the whole test set. On top, Absolute Percentage Error (APE) is measured on the code transformations of each program and the results are plotted through a histogram. On bottom, APE is measured on all data points of the test set and the measured speedups are plotted against their APE. We can see that the error gets smaller as speedups approach 1 and gets higher as speedups get far from 1. Particularly, the error is more significant for speedups below 0.05. The model is more accurate around speedup 1 because most programs in the training data set have speedups close to 1. Speedups below 0.05 are less frequent. The next experiment will evaluate whether the accuracy of the model allows finding the best code transformations when searching the space.

Search Space Exploration Using the Cost Model In this experiment, we evaluate the ability of search approach combined with the cost model to find good code transformation sequences for real-world benchmarks. We use BS and MCTS to explore the search space. We use a set of real-world benchmarks spanning different areas: image processing, deep learning, linear algebra and stencils. The benchmarks include *box blur* (an image processing filter to blur images), *conv + relu* (two successive neural network layers that benefit from operator fusion), *convolution* (a direct neural network convolution), *cvtcolor* (an image processing filter for converting the colors of an input image from RGB to gray), *doitgen* (a kernel from the multiresolution adaptive numerical scientific simulation (Louis-Noel, 2010)), *heat2d* (heat equation over 2D space), *heat3d* (heat equation over 3D space), *jacobi2d* (a jacobi-style stencil computation over 2D data with 5-point stencil pattern), *mvt* (matrix vector multiplication composed with another matrix vector multiplication but with transposed matrix), and *seidel2d* (Gauss-Seidel style stencil computation over 2D data with 9-point stencil pattern). The sizes of the input data for each benchmark is provided in the appendix.

Figure 6 shows the best speedups found for each benchmark. The baseline is the original program where the outermost loop is parallelized (no other code transformation is applied). The first column (blue), reports results obtained when beam

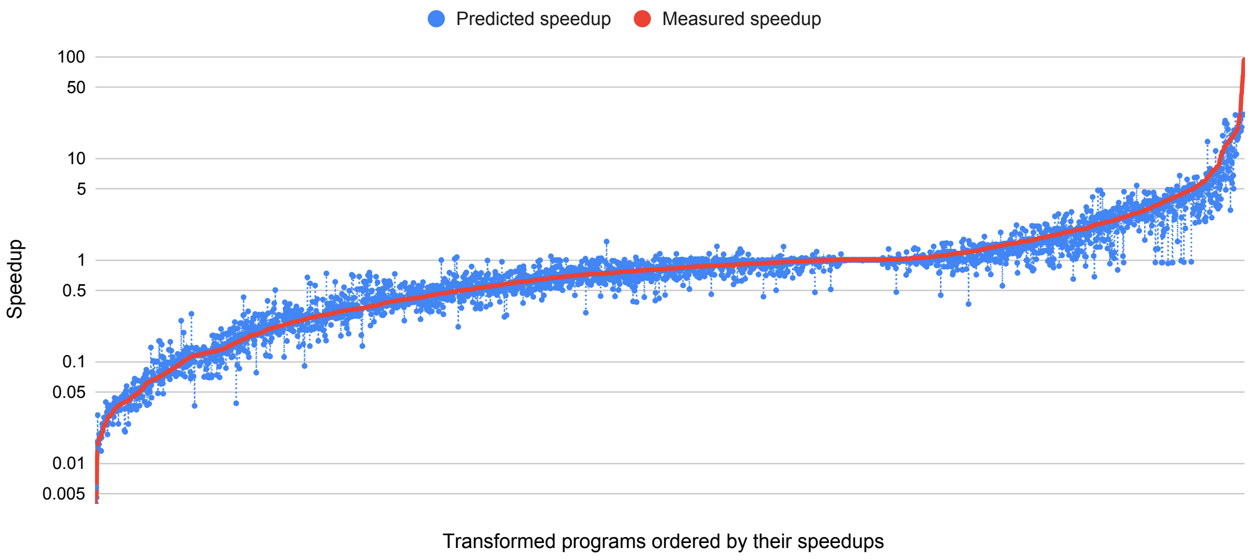


Figure 4: Predicted speedups compared to measured speedups. The speedups are ordered in ascending order.

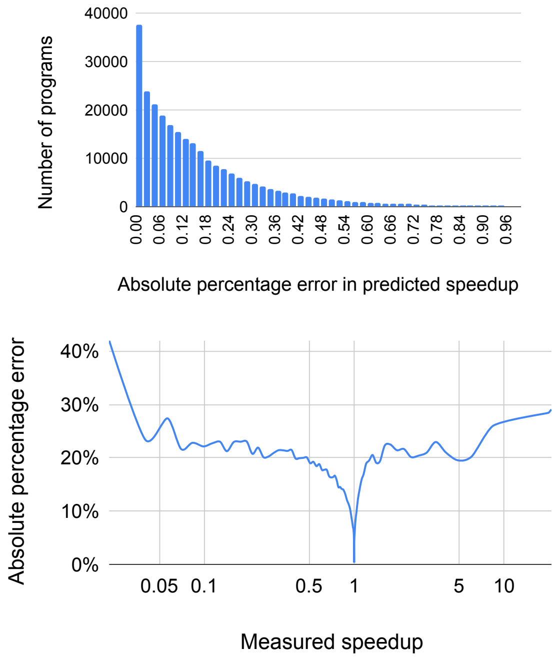


Figure 5: The distribution of error rates for the whole test set. On top, APE is measured for each transformed program, then the histogram of measurements is plotted. On bottom, APE is measured for each transformed program, then the speedups are plotted with their APE.

search is used to explore the search space. This column is considered the reference in our comparison as execution is used to obtain the speedups. In the second and third columns, beam search and MCTS use the cost model to predict speedups. The last column shows the speedups obtained after applying the Halide autoscheduler (Halide automatic optimizer) defined in (Adams et al., 2019).

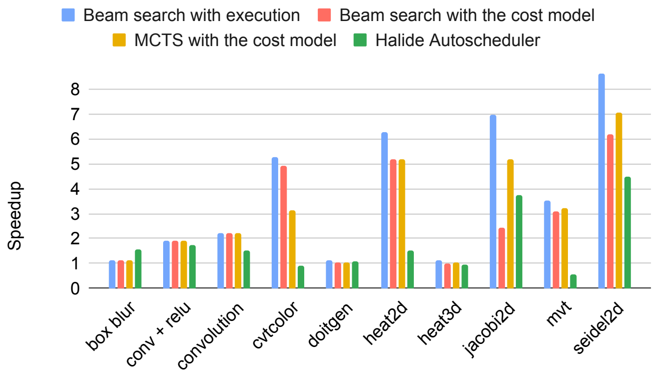


Figure 6: Speedups for different benchmarks obtained by exploring the search space.

Beam search (BS) with the cost model is competitive in most benchmarks, but does not find the best code transformations in *heat2d*, *jacobi2d* and *seidel2d*. Beam search with the cost model relies entirely on predictions to make decisions. Bad predictions can thus mislead the search method which is why beam search does not find the best transformations in the previous benchmarks. MCTS has similar performance, except in *jacobi2d* and *seidel2d* where it finds better code transformations, and in *cvtcolor* where the code transformations found are less good. MCTS can find better code transformations in these cases because it copes with model imprecision taking into account its stochasticity. However, since the tree space is explored differently, MCTS might explore different nodes compared to BS and thus have distinguishable results.

Comparison with Halide In this section, we compare our cost model with the one of Halide (Adams et al., 2019), a state-of-the-art cost model and the closest to ours. In comparison with Halide, TIRAMISU finds transformation sequences that are either competitive with those found by Halide or better (except in *box blur*). This is mainly due to miss predictions by the Halide model which lead Halide to use transformations that degrade performance. These wrong predictions happen in particular in benchmarks that are from the area of scientific computing which Halide was not trained to handle (*heat2d*, *jacobi2d*, *mvt* and *seidel2d*). In benchmarks that fall in the categories of deep learning and image processing, which Halide supports well, TIRAMISU and Halide have comparable performance.

We also compare the performance of the Halide model with that of TIRAMISU on randomly generated programs. Halide's paper uses R^2 as an accuracy metric and uses MSE (Mean Square Error) as a loss function, we thus use the same metric and loss function for comparison. Halide has an R^2 of 0.96, whereas TIRAMISU has 0.89. Both Halide and TIRAMISU have comparable results but Halide uses heavy feature engineering. The main advantage of TIRAMISU is that it does not require feature engineering.

Tradeoff Between Search Time and Quality of Code Transformations It is crucial to note that the execution time of a search method is dependent on the time to evaluate code transformations. If one compiles and executes every transformed program to assess the speedup of code transformations, the search time would increase considerably and therefore becomes impractical. The proposed model accelerates the search space exploration, reducing thus the time needed to find the best code transformations.

Table 2 illustrates the tradeoff that we make between the time needed to explore the search space, and the performance of the final code transformations found. It compares the gains from reducing the search time and performance degradation due to the use of a model. To be specific, for each benchmark: (1) we compare the time necessary to explore the search space using *Beam Search with Execution* (*BSE*) and using either of *Beam Search with the Cost Model* (*BSM*) on the left table, or *MCTS* on the right table. Comparing these two shows how faster *BSM* and *MCTS* are with regard to *BSE*. (2) we compare the performance of the final code transformations returned by *BSE* and the ones returned by *BSM* or *MCTS*.

The second column of the table represents the speedup in search time (the time taken by *BSE* over the time taken by *BSM* or *MCTS*). The third column is the degradation in performance (execution time) of the code transformations found by *BSM* or *MCTS* compared to the code transformations found by *BSE*. We can see that, on average, searching with *BSM* is 106.5 times faster than searching with *BSE*, while incurring an average decrease of 15% in the performance of the code transformations found. Moreover, searching with *MCTS* is 11.8 faster on average, with a loss of 12.5% in performance.

Note that for *jacobi2d* in beam search, and *cvtcolor* in MCTS, the performance degradation is important. This is mainly due to the imprecision of the model for these benchmarks. The model predicts some bad code transformations as being good. This is mainly because a pattern that appears in those benchmarks is not covered enough by the training data. A solution to this problem would be to generate more data that include the missing pattern.

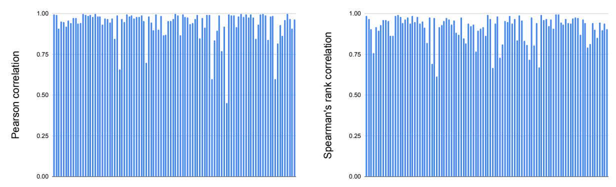


Figure 7: Correlation between predicted and measured speedups for the data points in the *small* test set. Pearson correlation is on the left and Spearman's correlation is on the right. Each column represents the coefficient measured on 32 random code transformations for a single program.

More Detailed Evaluation Figure 7 gives more insights on the behavior of our model. The Pearson and Spearman's coefficients are measured on a subset of the test set between the code transformations of each program. To simplify visualization, we use the same subset of transformations as the experiment of Figure 4. In most cases, the coefficients are close to 1, highlighting a strong linear correlation between the predicted and measured speedups (Pearson correlation), and between the predicted and measured ranks (Spearman's correlation). Particularly, the correlation between ranks is of most importance for search methods as these methods rely on the ranks of the explored points instead of their actual evaluations.

7 Related Work

Several machine learning based cost models have been developed for automatic code optimization. MILEPOST GCC (Fursin et al., 2008) uses a 1-nearest-neighbor model that takes manually engineered features and predicts the best combinations of compiler flags for GCC (GNU Compiler Collection). Ithemal (Mendis et al., 2018) uses an LSTM based model to predict the throughput of basic blocks of the control-flow graph (assembly-level code). Both of MILEPOST GCC and Ithemal do not support high-level loop transformations though. Modeling loop transformations is in particular challenging as it requires the ability to model the loop structure. This implies the ability to model cycles in the control-flow graph of the program, which is not trivial. In addition, Ithemal does not model memory accesses to different memory hierarchies (it only models accesses to cache). Unlike both systems, our proposed model supports

Benchmark	Search time improvement (speedup)	Performance degradation for code transformations
box blur	65x	0 %
conv + relu	12x	0 %
convolution	9x	0 %
cvtcolor	65x	7 %
doitgen	114x	7 %
heat2d	117x	18 %
heat3d	351x	13 %
jacobi2d	122x	65 %
mvt	83x	12 %
seidel2d	127x	28 %
Average	106.5x	15 %

Benchmark	Search time improvement (speedup)	Performance degradation for code transformations
box blur	19x	0 %
conv + relu	14x	0 %
convolution	11x	0 %
cvtcolor	3x	41 %
doitgen	12x	5 %
heat2d	17x	17 %
heat3d	28x	9 %
jacobi2d	4x	26 %
mvt	5x	9 %
seidel2d	5x	18 %
Average	11.8x	12.5 %

Table 2: Search time improvement compared to performance degradation. On the left, the results for beam search with the cost model are shown, and on the right, the results for MCTS.

loop transformations and takes into consideration different memory hierarchy levels.

Other related work includes a model proposed by Rahman et al. (Rahman et al., 2010). This model uses a feedforward neural network to predict the execution time of programs after applying the tiling code transformation. Another model proposed by Magni et al. (Magni et al., 2014) uses a feedforward neural network in a cascade fashion to predict the best thread-coarsening factor. These two methods rely on hand-engineered features though. They are also limited to a single code transformation.

Halide (Adams et al., 2019) proposes a more comprehensive method to find efficient code transformations. It combines beam search with a feedforward neural network that predicts the execution time of programs from a set of manually-engineered features. It uses 54 heavily engineered features to perform its predictions. To the best of our knowledge, both of Tiramisu and Halide consider the same code transformations in their search spaces. In the same context, AutoTVM (Chen et al., 2018) uses a deep learning model to search for code transformation parameters (TVM compares two models, a TreeGRU and a Gradient Boosted Tree). The TVM models are used with simulated annealing to search the space. The authors only demonstrate the search for transformation parameters though (tile size, unrolling factor, ...). They do not demonstrate search for loop transformations since the transformations themselves are provided by the user. In a different fashion, DeepTune (Cummins et al., 2017) proposes a neural network consisting of embedding layers and LSTM cells to predict whether an OpenCL kernel should be mapped to CPU or GPU. DeepTune also proposes another model to predict whether thread coarsening (a code transformation for GPUs) should be used. The DeepTune models are classification models though. This means that they can classify whether a given transformation is beneficial, but they are not designed to rank different sequences of code transformations. Therefore, they are not ideal for use when searching a large space of code transformations where many sequences need to be ranked and the best one selected.

The goal of this paper is not to propose a cost model for general purpose compilers. The goal is also not to propose a cost model for general purpose transformations. In a way similar to related work, this paper mainly focuses on a domain specific compiler and on loop transformations.

Many polyhedral compilers including Pluto (Bondhugula et al., 2008), PENCIL (Baghdadi et al., 2015a; 2013a), LLVM Poly (Grosser et al., 2012), and Tensor Comprehensions (Vasilache et al., 2018) formalize the problem of automatic code optimization as an integer linear program (ILP). The objective of this ILP is to minimize the distance between producer and consumer statements. The resulting problem can be solved exactly, but the implicit cost model does not capture all the complexity of the hardware architecture and transformation interactions (Baghdadi et al., 2019). This leads to suboptimal solutions (Baghdadi et al., 2019; 2015a; 2013b). Making the objective function more comprehensive makes the problem non-linear and thus it becomes intractable.

8 Conclusion

This paper presents a novel cost model for predicting speedups. This cost model is a *regression* cost model that operates on *full programs* and *does not rely on extracting complex features*. It is not limited to transformation parameters but also includes code transformations. We develop a random code generator to generate the training data and release the generator publicly. We evaluated the proposed model and show that it had a low error rate of 16% MAPE. We integrate this model in a search space method and show that the integrated approach enables TIRAMISU to automatically find sequences of code transformations that are competitive with state of the art compilers.

REFERENCES

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL http://doi.acm.org/10.1145/3306346.3322967.

Bachir, M., Brault, F., Gregg, D., Cohen, A., et al. Minimal unroll factor for code generation of software pipelining. *International Journal of Parallel Programming*, 41(1): 1–58, 2013.

Baghdadi, R., Cohen, A., Guelton, S., Verdoolaege, S., Inoue, J., Grosser, T., Kouveli, G., Kravets, A., Lokhmotov, A., Nugteren, C., Waters, F., and Donaldson, A. F. PENCIL: towards a platform-neutral compute intermediate language for dsls. *CoRR*, abs/1302.5586, 2013a. URL http://arxiv.org/abs/1302.5586.

Baghdadi, R., Cohen, A., Verdoolaege, S., and Trifunovic, K. Improved loop tiling based on the removal of spurious false dependences. *TACO*, 9(4):52, 2013b.

Baghdadi, R., Beaugnon, U., Cohen, A., Grosser, T., Kruse, M., Reddy, C., Verdoolaege, S., Betts, A., Donaldson, A. F., Ketema, J., Absar, J., Haastregt, S. v., Kravets, A., Lokhmotov, A., David, R., and Hajiyev, E. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pp. 138–149, Washington, DC, USA, 2015a. IEEE Computer Society. ISBN 978-1-4673-9524-3. doi: 10.1109/PACT.2015.17. URL http://dx.doi.org/10.1109/PACT.2015.17.

Baghdadi, R., Cohen, A., Grosser, T., Verdoolaege, S., Lokhmotov, A., Absar, J., van Haastregt, S., Kravets, A., and Donaldson, A. F. PENCIL language specification. Research Rep. RR-8706, INRIA, 2015b. URL https://hal.inria.fr/hal-01154812.

Baghdadi, R., Ray, J., Romdhane, M. B., Del Sozzo, E., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pp. 193–205, Piscataway, NJ, USA, 2019. IEEE Press. ISBN 978-1-7281-1436-1. URL http://dl.acm.org/citation.cfm?id=3314872.3314896.

Baghdadi, R., Debbagh, A. N., Abdous, K., Benhamida, F. Z., Renda, A., Frankle, J. E., Carbin, M., and Amarasinghe, S. Tiramisu: A polyhedral compiler for dense and sparse deep learning, 2020.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*, pp. 101–113, 2008.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pp. 3389–3400, 2018.

Cummins, C., Petoumenos, P., Wang, Z., and Leather, H. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232. IEEE, 2017.

Feautrier, P. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*, pp. 429–441, St. Malo, France, 1988. ACM. ISBN 0-89791-272-1. doi: 10.1145/55364.55406. URL http://portal.acm.org/citation.cfm?id=55406.

Fursin, G., Miranda, C., Temam, O., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M., Barnard, P., Ashton, E., Courtois, E., and Bodin, F. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada, June 2008. URL https://hal.inria.fr/inria-00294704.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9: 249–256, 01 2010.

Grosser, T., Groslinger, A., and Lengauer, C. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012. URL http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12.

Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pp. 66:66–66:75, New York, NY, USA, 2014. ACM.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Lefebvre, V. and Feautrier, P. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998. ISSN 01678191. doi: 10.1016/S0167-8191(98)00029-5.

Loshchilov, I. and Hutter, F. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017. URL http://arxiv.org/abs/1711.05101.

Louis-Noel, P. PolyBench suite. http://www.cse.ohio-state.edu/~pouchet/software/polybench/, 2010. URL http://www.cse.ohio-state.edu/~pouchet/software/polybench/.

Magni, A., Dubach, C., and O'Boyle, M. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pp. 455–466, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328098. doi: 10.1145/2628071.2628087. URL https://doi.org/10.1145/2628071.2628087.

Mendis, C., Amarasinghe, S. P., and Carbin, M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. *CoRR*, abs/1808.07412, 2018. URL http://arxiv.org/abs/1808.07412.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Paul, F. and Christian, L. The polyhedron model. In Padua, D. (ed.), *Encyclopedia of Parallel Computing*, pp. 1581, 1592. Springer, 2011.

Quilleré, F. and Rajopadhye, S. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems*, 22(5):773–815, September 2000.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012. ISSN 0730-0301.

Rahman, M., Pouchet, L.-N., and Sadayappan, P. Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag*, 2010.

Smith, L. N. and Topin, N. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR*, abs/1708.07120, 2017. URL http://arxiv.org/abs/1708.07120.

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

Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjodin, J., and Upadrasta, R. GRAPHITE two years after: First lessons learned from Real-World polyhedral compilation, January 2010.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

Wolf, M. E. and Lam, M. S. A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on parallel and distributed systems*, 2(4):452–471, 1991.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237.

A Appendix

A.1 Model Architecture Details and Training Methodology

The computation embedding layer is a fully connected multilayer perceptron (MLP), feedforward neural network that takes 1235-dimensional computation vectors and generates 180-dimensional embeddings. We use 3 intermediate layers of 600, 350, 200 neurons respectively. The output of each layer is transformed by the *ELU* function and fed to a dropout layer with a dropout probability of 0.225, and then passed to the next layer. This succession of the activation function and the dropout layer is applied to all the neural networks of this model. The two LSTM cells in the loop embedding unit have identical input and hidden vector sizes that correspond to the output of the computation embedding layer (180). The feedforward neural network that maps the concatenated hidden vectors to 180-dimensional loop embedding have one intermediate layer of size 200. The regression layer that maps the *program embedding* vector to a speedup value, has two intermediate layers with 200 and 180 neurons.

We implemented our model in PyTorch (Paszke et al., 2019) (0.4.1.post2). All model parameters are learnable from the computation embedding layer to the regression layer by way of the recursive loop embedding layer. For the loss function, we used MAPE, a normalized metric based on L_1. This loss function is suitable for speedup prediction because the target value is positive by design. In addition, the function motivates the model to be equitably accurate in the wide range of speedups we have. The Glorot initialization (Glorot & Bengio, 2010) is adopted for all weights of the model. We train our model using AdamW (Loshchilov & Hutter, 2017) with a weight decay coefficient of 0.0075. The learning rate is scheduled by the One Cycle Policy (Smith & Topin, 2017) with a maximum learning rate of 0.001. The other optimizer parameters are left on their default values. The best accuracy is achieved after about 700 epochs of training. The training set is processed in batches of 32 data points. Each batch is formed by code transformations belonging to the same algorithm. The rationale for this grouping is that it is faster to operate on data points having the same tree structure.

A.2 Benchmark Sizes and Parameters

Table 3 summarizes the benchmarks' input sizes and parameters used in Section 6.

A.3 More Detailed Evaluation

Figure 8 gives an overview of the correlation between the predicted and measured speedups over 16 programs randomly selected from the test set. Each chart represents the 32 random transformations applied on each program with blue dots, the closer a blue dot is to the red line the lower the prediction error is. This figure shows that the cost model's predictions fit well the distribution and the range of the speedups and does not just predict an average value for each program.

Benchmark	Input size and parameters
box blur	$3 \times 1024 \times 1024$
conv + relu	batch size: 8 input size: $1024 \times 1024 \times 3$ kernel size: 3×3 output features: 2
convolution	batch size: 8 input size: $1024 \times 1024 \times 3$ kernel size: 3×3 output features: 2
cvtcolor	$3 \times 1024 \times 1024$
doitgen	$256 \times 256 \times 128,\ 256 \times 256$
heat2d	1024×1024
heat3d	$770 \times 898 \times 1024$
jacobi2d	130×1024
mvt	1024×1024
seidel2d	256×256

Table 3: Benchmarks input sizes and parameters

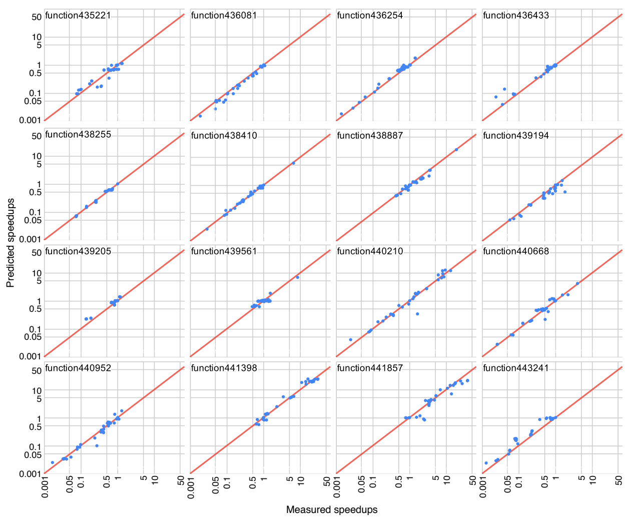


Figure 8: Measured vs predicted speedup on 16 random programs from the test set, each blue dot represents a code transformation with respect to its measured speedup and its predicted speedup.