Tiling parameters prediction using Machine Learning techniques

Maksim Berezov, Corinne Ancourt, Mikhail Kashchenko

Tiling parameters prediction using Machine Learning techniques

Maksim Berezov[1], Corinne Ancourt[1], and Mikhail Kashchenko[2]

[1] MINES Paris, Paris, France {maksim.berezov, corinne.ancourt}@mines-paris.fr
[2] Independent researcher, Berlin not4lettersde@gmail.com

Abstract. The tiling transformation is one of the most crucial code optimization techniques to expose data locality and parallelism. The main idea is to split the initial iteration space into blocks and traverse them in a special order. This transformation is parametric and very sensitive to parameter tuning. Poor parameter tuning can lead to much lower performance than the initial code. Existing state-of-the-art solutions consider a restricted list of parameters to handle this issue and guarantee safe solutions.

Our work proposes solutions that go beyond current state-of-the-art techniques and gain additional speedup considering a larger set of options for tiling. Our approach is based on Machine Learning methods and automatically derives heuristics to tune tiling parameters. We can predict: 1) the optimal partitioning matrix of the iteration space 2) the tile sizes, 3) the optimal directions for scanning inter-tiles, 4) the optimal directions for scanning intra-tile elements. The optimal selection of these parameters is crucial especially for programs that have data dependencies.

We introduce sets of features that feed our models in their predictions. The first set encodes data dependencies, the second one captures the level of parallelism and data locality in a code. The third one aggregates information about the iteration space.

Our approach surpasses existing feature spaces for tiling parameters prediction. Moreover, it could be used in conjunction with auto-tuners to iterate through the iterative search.

Keywords: code optimization, compilation, source-to-source transformations, machine learning, loop tiling, data-dependencies, feature space design

1 Introduction

The loop tiling transformation is considered to be one of the more complex code transformations. Its application can provide significant execution time gains because it allows targeting different memory levels including cache levels. Loop tiling depends on data dependencies and impacts many code aspects such as the temporal and spatial locality of data, the parallelism.

Many factors explain the complexity of predicting the tiling parameters including the large space of possible solutions. Typically, researchers only predict the sizes of rectangular tiled shapes. In addition to tile sizes, our work investigates the profitability of choosing different tile shapes and tile scanning directions for code generation. These options are poorly reflected in existing state-of-the-art solutions, but can potentially provide significant performance gains. We classify the methods of parameter selection for code transformations as:

- Analytical models derived by an expert [5], [6], [22], [23] [13], [17], [10], [28], [8]
- Iterative auto-tuners [24], [25], [19], [9], [2], [20], [12]
- Static analytical models derived by Machine Learning algorithms [29], [15], [21], [16]

Analytical models derived by experts provide correct solutions but are more difficult to implement than the others because they must be adapted very precisely for each parameters to the target machine. Increasing the set of parameters degrades the quality and stability of these solutions. This is also the case with auto-tuners, moreover, the search time increases in proportion to the size of search parameters. Our study focuses on the third approach.

This paper is organized as follows. Section 2 introduces the context of tiling transformation and its parameters. Section 3 presents the proposed features that would be used for ML modelling. Section 4 highlights our experiments in Machine Learning modelling. It contains steps like synthetic data generation, model training and model evaluation.

2 Loop Tiling Parameters

The potential performance that you can get by applying loop tiling depends on the choice of the loop tiling parameters but also on the quality of the code generated after tiling. Indeed, once the tile parameters have been chosen, there are several possible execution paths to generate the tiled code which respect the semantics of the initial program. The questions of the choice of the directions for scanning the tiles and the ordering of the computational iterations in the tiles arise.

This section introduces important criteria: parameters and code generation options that have an impact on the loop tiling transformation performance.

2.1 Kernels of interest

Our methodology allows optimizing kernels with any kind of loops.

But the highest performance gain could be achieved for kernels with constant data dependencies. It is intuitive since some options to optimize tiling parameters rely on the fact that dependencies exist. Hence, for this case, we have a more restricted set of optimizations and less potential benefit.

Thus, this paper takes place in the context of kernels with constant data dependencies. The methodology could be applied to a kernel of any kind. But we conclude that more effective code is generated under this context.

2.2 Scanning directions for tiles and its elements

The loop tiling transformation implies the partitioning of the iteration space into blocks according to the chosen partitioning matrix [11]. However, the order in which the blocks are traversed is not unique. Data dependencies define the correctness of execution (block traversal). Likewise, the order of traversal of the points inside the block is not unique. Since several possibilities are possible, we can consider this choice as a parameter to predict.

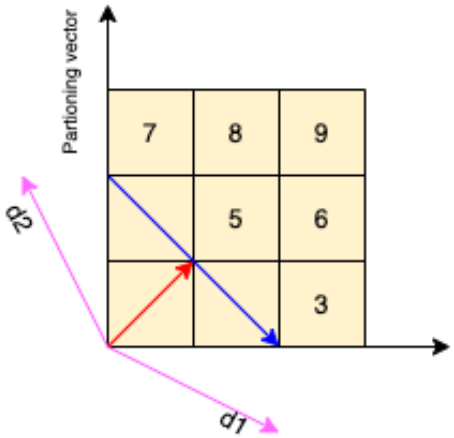

Fig. 1: Inter-tile scanning

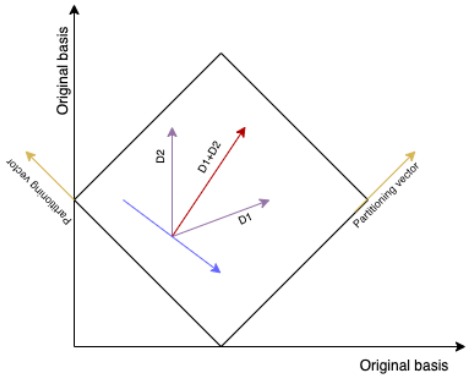

Fig. 2: Intra-tile scanning

PIPS compiler [18] offers nine possible choices for generating tiled code. It is a combination of the three possible choices for scanning the tiles and the three for scanning the elements inside the tiles.

Figure 1 illustrates one possible way of scanning the tiles of the computational domain according to its data dependency vectors. Let us define d1 and d2 as the extreme rays of the dependence cone which summarizes the set of dependencies of the computational kernel [27]. The red vector represents their sum (the sequential hyperplane direction), the blue vector is perpendicular to the red vector and represents a potential parallel direction [14]. The red and blue vectors describe a possible basis of scanning the tiles that we refer to as $\mathtt{T}_{Parallel}$ tile direction. The sequential hyperplane direction carries all dependencies, while blocks along the parallel direction can be executed concurrently. So, the sequence of blocks 1-4-

2-7-5-3-8-6-9 represents a legal execution if we apply $\mathtt{T}_{Parallel}$ scanning; among others blocks 3-5-7 can be run in parallel.

The second possible choice for the directions scanning tiles is named $\mathtt{T}_{Shape}$. Its scanning directions are parallel to the partitioning vectors of the tiling shape (rectangular shape in this case). Thus the sequence of blocks 1-2-3-4-5-6-7-8-9 is an example of legal execution in this case. The third possible scanning direction $\mathtt{T}_{Initial}$ corresponds to the initial iteration space and matches the second case for a rectangular shape.

We use the same strategy to define possible scanning directions for the elements inside each tile:

- $\mathtt{L}_{Initial}$: relative to the initial basis,
- $\mathtt{L}_{Parallel}$: corresponding to the hyperplane sequential direction and its relative orthogonal vector,
- $\mathtt{L}_{Shape}$: relative to the partitioning vector directions.

Figure 2 illustrates scanning directions for inter tile. The same idea of possible scanning vectors is applied here.

In the general case, we have nine combinations of scanning directions. However, some of them may not be legal due to data dependencies.

2.3 Data Dependence Abstraction for Tiling

The second concept that we would like to introduce is the abstraction used to encode dependencies. Yang et al. [27] have defined the minimal abstraction for a loop transformation. The dependence cone turned out to be the minimal abstraction to legally apply the tiling transformation, which means that among the studied abstractions, it is the smallest abstraction which is enough precise to check whether the transformation is legal. In our study, we encode this abstraction to a feature vector for Machine Learning issues. The dependence cone is the convex hull of the set of points that are positive linear combination of dependence distance vectors. More formally, the dependence cone is defined as $DC(L) = \{v = \sum_{i=1}^{k} \lambda_i d_i \in Z^n | d_i \in D(L), \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i \geq 1\}$, where $D(L)$ is the set of all distance vectors d_i in loop nest L.

Figure 3 shows an example of dependence cone for a code with d1, d2, d3 dependence vectors. Note, that vectors d3 and d1 form the extreme rays of this cone.

2.4 Tiling sizes

The right choice of the tiling sizes is a crucial component of the tiling transformation. Large tile sizes can result in a large number of slow high-level caches uses. In contrast small tiles may not fully benefit from improved data locality.

The selection of tile size is a complex problem that depends on many criteria such as the application, the architecture, the cache hierarchy. In our study, we only vary the application characteristics such as the size of the iteration domain, the access functions for array elements, and the data dependencies. The architecture is fixed.

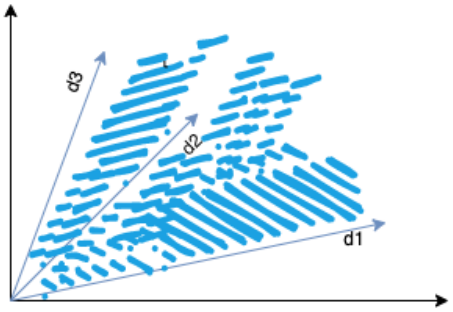

Fig. 3: Dependence cone

2.5 Parallel code generation

After applying tiling and exploiting parallelism in the nest of loops, several loops can be parallelized. In OpenMP, it is not beneficial to generate several levels of parallel loops because the creation of new parallel threads in threads has an additional cost. It is more efficient to keep a single parallel loop externally and a vector loop internally.

The choice of which parallel loop to keep when there are several possible is tricky. We must take into account the number of iterations of these loops; it must be greater than the number `np` of processors and be a multiple of it, if possible, to take full advantage of the potential number of architectural parallelism. The outermost loop should be preferred for GPU or coarse grain architectures.

In the framework of this study, we exploit the parallelism of loops by generating OpenMP parallel directives for the outermost parallel loop and OpenMP vectorial directives for the innermost loop if it is parallel.

The number of threads has been chosen to use the maximum number of parallel cores available and multi-threading when possible.

3 Feature space design

Machine Learning techniques draw conclusions based on some characteristics of a phenomenon under consideration. Usually, it is a vector of fixed size, and each value represents relevant properties of the phenomenon [4]. A choice of irrelevant features implies that the characteristics of the phenomenon are not sufficiently represented. Machine Learning algorithms cannot generalize based on them. It would lead to unsatisfactory results and poor performance. Therefore, the design of a suitable function space comes to the fore.

We distinguish two main concepts which are crucial for the selection of tiling parameters. These are 1) the data dependencies, 2) the array access functions, and information about the iteration domain. Data dependencies define the legality of the tiling and motivate the use of different scanning directions [1].

Array access functions associated with the iteration domain characterize the spatial and temporal data locality and the parallelization/vectorization opportunities [29], [15]. Array access functions are important criteria to choose the correct tile sizes.

3.1 Encoding of dependencies

Encoding dependencies is not an easy task. The main problem is that although the concept of data dependencies is well-defined, they can be represented or approximated in many ways. Moreover, the information about data dependencies needs to be transferred to a vector of a fixed size for ML purposes. To find the proper level of abstraction, we investigate research about the suitable abstractions for data dependencies to apply tiling.

Abstractions to encode data dependencies in our ML model We use three abstractions to encode the data dependency information. We illustrate them in the example of Listing 1.1.

Listing 1.1: Original code

```
1 for (int i = 2; i < 1022; i++)
2   for (int j = 2; j < 510; j++)
3     A[i][j] =  A[i-2][j-1] + A[i-1][j-1] + A[i-1][j-2];
```

1 - Dependence cone The dependence cone is the minimal abstraction to verify the legality of the loop tiling application. The dependence cone can be represented as a set of extreme rays. The first abstraction encodes the areas of the iteration space where we observe these extreme rays. For each extreme ray, we compute its angle with the x-axis in degrees. Here, x represents the first dimension of the dependency vector.

Our abstraction to encode the information on the extreme rays is a vector of fixed size (8 for 2-dimensional iteration domain). Let's describe the elements of the vector from 0 to 7. The i-th element of this vector gives how many extreme rays of the dependence cone form the angle θ with the x-axis, such as $\theta \in [45\times$ i; $45 \times$ (i+1)).

The mapping function for 3-dimensional iteration domains is pretty similar, except that it goes through the coordinates of two polar angles for the dependency vectors instead of one. The vector summing these types of vectors is of size 32 (c.f. section 3.3).

2 - Summation vector The second abstraction related to the data dependencies is a vector which is the sum of all data dependency vectors. We extract a unique value - the angle of this vector with the x-axis. This information is required because this vector is used as a basis vector for some scanning directions ($\mathtt{T}_{Parallel}$ and $\mathtt{L}_{Parallel}$ directions). In the example 1.1, the sum of three dependence vectors results in vector (4,4), which makes an angle of 45 degrees with the x-axis.

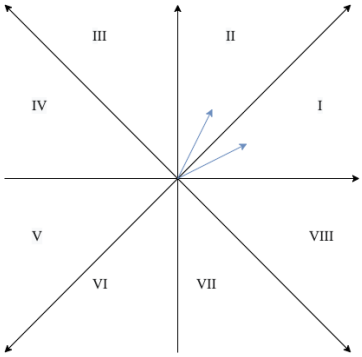

Fig. 4: Encoding of the extreme rays

3 - Encoding of uniform data-dependencies In the same way, we encode the extreme rays 3.1, we encode the uniform data dependence vectors. Our abstraction to encode the information on the uniform dependencies is a vector of fixed size (8 for 2 dimension iteration domain). The i-th element of this vector gives how many uniform data dependencies form the angle θ with the x-axis, such as $\theta \in [45\times$ i; $45 \times$ (i+1)).

This vector for listing 1.1 equals [1,2,0,0,0,0,0,0].

3.2 Encoding of the iteration domain

The iteration domain can have an impact on the prediction of all the parameters considered. We propose to encode it with two simple features: 1) the number of iterations for each loop and their total 2) the presence of constant loop bounds for all loops.

The first feature vector is a vector of the size that equals the number of nested loops + 1. The first value represents the number of iterations for the first nested loop (NaN, if it is unknown, e.g. variables of functions), the second value represents the number of iterations for the second nested loop, etc. The last value represents the total number of iterations for the loop nest. This vector for listing 1.1 equals [1020, 508, 518160].

The second feature vector is a vector of the size that equals the number of nested loops. The first value represents the presence of constant loop bound (1) or their absence (0) for the first nested loop, etc. This vector for listing 1.1 equals [1,1] since the loop bounds are constant.

3.3 Generalization for 3-D case

In the proposed encoding of the dependence cone, data dependencies can be easily generalized for 3-D tiling and for higher dimensions. Let θ be the angle between a data-dependence vector/extreme ray of a dependence cone and the Z-axis. And ϕ is the angle between the projection of the same data-dependence

vector/extreme ray to the XY plane and X-axis. Let V denotes the vector, which contains all the subareas of 3-D space to be encoded. V = $\langle v_1\ v_2\ ...\ v_K \rangle$

where v_i is a particular subarea of 3-D space, $i \leq K$, K is the number of chosen subareas of 3-D space (e.g. we have experimented that 32 is a good choice). Then we can define the mapping function that assigns the subarea of the 3-D space to each value of θ and ϕ angles. Actually, this mapping function could be considered as the mapping from polar coordinates to the subareas of 3-D space.

$f(x) :< \phi, \theta > \rightarrow v_i \in V$

In the 3-D case, we need also two angles θ and ϕ to encode the information about the summation vector.

3.4 Encoding of array accesses

There is a bunch of research investigating the proper encoding of array accesses.

Yuki et al. [29] investigate the problem of automatic tiling selection using machine learning approaches. The authors consider cubic tiling on three nested loops with 2D data. They describe each loop according to the references inside the arrays. Liu et al. [15] propose a slightly similar approach. In the context of this paper, we use the concatenation of two vectors to properly encode the array accesses.

4 Machine Learning modeling

Our methodology is presented in Figure 5. It consists of four main steps: feature extraction, synthetic data generation, data labeling, machine learning pipeline and code generation process.

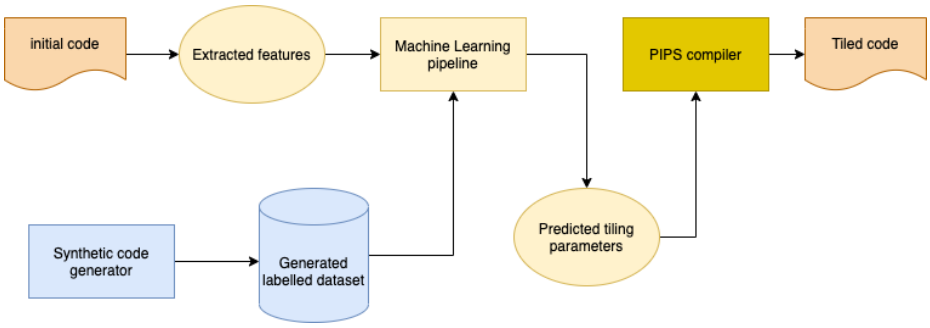

Fig. 5: Methodology

4.1 Data collection

We used the code generator [3] to generate around 1100 synthetic kernels. The main concept that we targeted during the generation was the creation of uniform data dependencies. The generated programs had from 1 to 5 true uniform data dependencies (constant write-before-read dependencies). We tried to generate kernels with the most different possible dependence cones. The methodology was to generate kernels whose second outermost loop could be parallel after the tiling transformation.

Autotuning process We used LOCUS Autotuner [24] to label the data for two different settings. First, we asked the Autotuner to find the optimal combination of parameters for the square tiling considering 4 different scanning directions: TI-LI, TI-LP, TP-LI, and TP-LP. TS and LS scanning directions are respectively identical to TI and LI in the case of square tiling. Second, we asked the Autotuner to tune the same kernel with diamond tiling considering 9 possible scanning directions. Scanning directions related to the partitioning shape (TS, LS) have been added for -Intra and -inter tiles. That gives 9 combinations in total. All the collected executions and corresponding execution times were collected for the learning process. Note that not all of these 9 combinations could be legal, so we check the legality and skip not legal executions.

4.2 ML modelling

We have an imbalance in our classes. The original tiling shape and scanning are the best options in almost 80% of cases. Hence, we think it reasonable to create a two-stage ML model that would consist of two parts. The first model will predict whether we consider only the original tile settings or the extended settings, the best option for the given code. The second model will predict the tile sizes if the answer is positive (not profitable for extended settings), or all the parameters for a given kernel (tiling shape+scanning directions+ tile size) otherwise. The pipeline is illustrated in Figure 6.

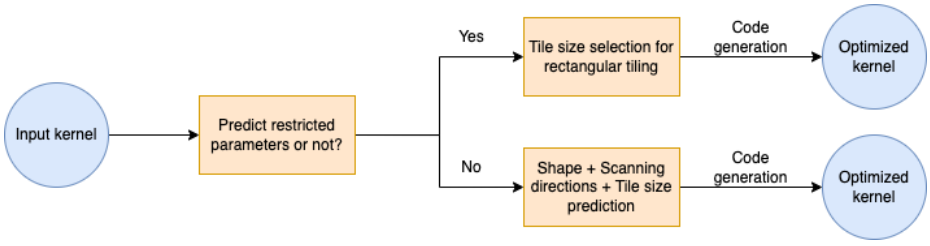

Fig. 6: Prediction pipeline

Our first-step model is a binary classification model. It takes the input features presented in section 3 and predicts whether this kernel is potentially a

good candidate for restricted parameters of tiling or not (extended parameters). We do modeling using the stacked classifier. We use Logistic regression [26] as a meta-classifier to aggregate the predictions.

Train/test split We evaluated the model on the test set. These kernels were not involved in the training phase. Kernels from the train set were involved in the training of all models in this subsection. The train/Test split ratio is 0.85/0.15 with the same class balance of the minority class in both splits.

Model evaluation We face a problem of imbalanced classification. We consider it reasonable to use the ROC-AUC score to evaluate the quality of the predictions. The ROC-AUC curve is a graph showing the performance of a classification model at all classification thresholds. We achieved a ROC-AUC score of 0.9, we conclude that our model has a good generalization ability to distinguish whether a kernel benefits from extended tiling parameters or not.

Second-step models We distinguish two models for the second step prediction. If the data sample was marked as beneficial for the extended tiling parameters then we predict the tile shape, scanning directions, and tile size. Otherwise, we only predict tile sizes. We consider it as a regression problem that predicts the speedup and we sample the best parameters that maximize the speedup for the prediction. We used single CatBoost [7] to handle this task.

Our approach to predict (tile sizes or shapes/scanning directions/tile size) is presented in Figure 7. The idea is to predict the speedup for a given code based on shapes/scanning directions/tile size and a feature vector. We sample all possible code parameters for a given kernel and predict the speedup for each parameter. We select the parameters with the highest predicted speedup. It should be noted that the speedup we predict cannot be used as an accurate estimate of the real speedup. There are many different factors that impact the speedup and we do not take them into account, but the predicted "speedup" helps differentiate the impact of the tiling parameters on real speedup. This approximation helps to choose the more appropriate combinations of parameters.

Evaluation of the results We propose the following method for the evaluation of our model. Firstly, we get predictions for the kernels from the test set that benefit from advanced tiling. Second, we generate optimized code according to the predicted tiling parameters. Finally, we measure the average absolute speedup of the mentioned above kernels.

We perform the same procedure for our two-stage pipeline and "simpler" ML model that predicts only cubic tile sizes. Both models were trained on the same dataset. The comparison of these two models helps us to understand how much performance gain we achieve if we increase the complexity of our model (predicted parameters). It is not always true that the increase in complexity leads to an increase in performance.

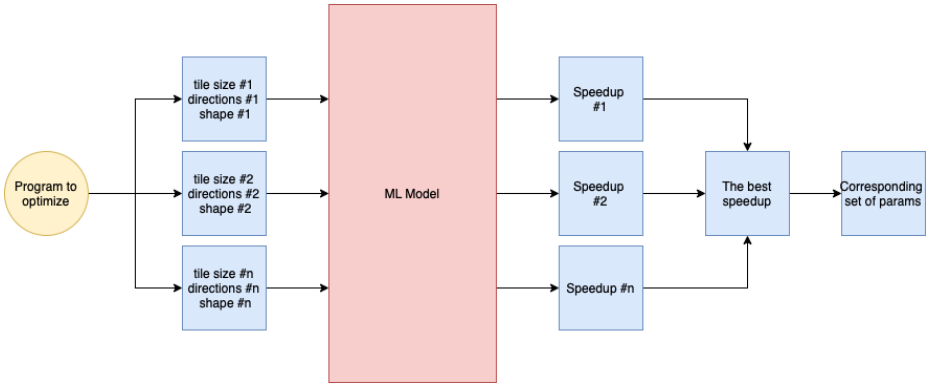

Fig. 7: Tile size prediction based on speedup prediction pipeline

Figure 8 shows the comparison of our approach and the "simpler" model. The average absolute speedup was normalized by the performance of the "simpler" model. This experiment shows that our methodology could bring up to 5% speedup for the kernels that benefit from it. The increase in model complexity leads to a gain in performance.

The relative speedup for these kernels is about 84% (out of the theoretical maximum). It is the ratio of the speedup that we archived with our method (one-shot prediction) to the speedup that was found by the LOCUS autotuner during the exhaustive search.

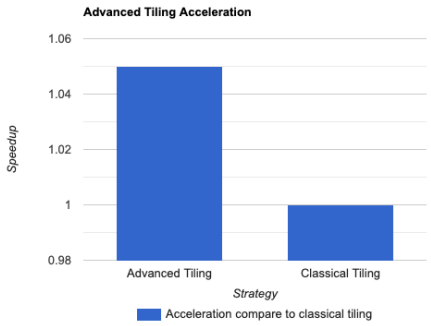

Fig. 8: Acceleration of classical pipeline

Experimental setup The experiments were run on Intel® Core™ i7-8650U 4C/4T @1.90GHz with capacity caches of L1: 32KB, L2: 256KB, L3: 8192KB and 32GB

DDR4 DIMM RAM, Phys. cores: 4, Compiler: GCC 5.4.0, Number of iterations per dimension in the test set: 1024, Number of Threads: 4, Opt. level: -O3

5 Concussion and Future Work

This paper presents a methodology to predict the optimal parameters for loop tiling transformation. We do not limit our predictions just with tile size but consider extended set of parameters. It helps to archive up to 5% of additional absolute two dimensional loops.

Our second contribution concerns the design of the feature space for the tiling prediction problem using Machine Learning. We propose different sets of features. They encode data dependencies, iteration domain and information about data locality and parallelism in the initial code. To our knowledge, our study is the first to use explicit information about data dependencies for tiling parameter prediction.

Our future work concerns the improvement of the training/validation sets. Our methodology supports the usage of tiling of arbitrary perfectly-nested loop-nests, but training data contains just 2-dimensional loops. Moreover, we used synthetically generated kernels for model evaluation. Collection of a large benchmark of real-worlds kernels would make our results more reliable. Also, we would like to perform our experiments on different architectures (GPU, other CPUs) and build the architecture-dependent model. It would use architecture-dependent features (e.g. memory hierarchy features) to be able to generalize across different architectures. The model presented does not benefit from these features because the experiments were performed on one architecture.

References

1. Ancourt, C., Irigoin, F.: Scanning polyhedra with do loops. In: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 39–50 (1991)
2. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.M., Amarasinghe, S.: Opentuner: An extensible framework for program autotuning. In: Proceedings of the 23rd international conference on Parallel architectures and compilation. pp. 303–316 (2014)
3. Berezov, M., Ancourt, C., Zawalska, J., Savchenko, M.: Cola-gen: Automatic code generator for program optimization using active learning techniques. CRI report **1**(1), 1–10 (2021)
4. Blum, A.L., Langley, P.: Selection of relevant features and examples in machine learning. Artificial intelligence **97**(1-2), 245–271 (1997)
5. Chame, J., Moon, S.: A tile selection algorithm for data locality and cache interference. In: Proceedings of the 13th international conference on Supercomputing. pp. 492–499 (1999)
6. Coleman, S., McKinley, K.S.: Tile size selection using cache organization and data layout. ACM SIGPLAN Notices **30**(6), 279–290 (1995)
7. Dorogush, A.V., Ershov, V., Gulin, A.: Catboost: gradient boosting with categorical features support. arXiv preprint arXiv:1810.11363 (2018)

8. Gysi, T., Grosser, T., Hoefler, T.: Absinthe: learning an analytical performance model to fuse and tile stencil codes in one shot. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 370–382. IEEE (2019)
9. Hoste, K., Eeckhout, L.: Cole: compiler optimization level exploration. In: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization. pp. 165–174 (2008)
10. Hsu, C.H., Kremer, U.: A quantitative analysis of tile size selection algorithms. The Journal of Supercomputing **27**(3), 279–294 (2004)
11. Irigoin, F., Triolet, R.: Supernode partitioning. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 319–329 (1988)
12. Knijnenburg, P.M., Kisuki, T., Gallivan, K., O'Boyle, M.F.: The effect of cache models on iterative compilation for combined tiling and unrolling. Concurrency and Computation: Practice and Experience **16**(2-3), 247–270 (2004)
13. Lam, M.D., Rothberg, E.E., Wolf, M.E.: The cache performance and optimizations of blocked algorithms. ACM SIGOPS Operating Systems Review **25**(Special Issue), 63–74 (1991)
14. Lamport, L.: The parallel execution of do loops. Communications of the ACM **17**(2), 83–93 (1974)
15. Liu, S., Cui, Y., Jiang, Q., Wang, Q., Wu, W.: An efficient tile size selection model based on machine learning. Journal of Parallel and Distributed Computing **121**, 27–41 (2018)
16. Malik, A.M.: Optimal tile size selection problem using machine learning. In: 2012 11th International Conference on Machine Learning and Applications. vol. 2, pp. 275–280. IEEE (2012)
17. Mehta, S., Beeraka, G., Yew, P.C.: Tile size selection revisited. ACM Transactions on Architecture and Code Optimization (TACO) **10**(4), 1–27 (2013)
18. ParisTech, M.: Pips: Automatic parallelizer and code transformation framework (2013)
19. Pouchet, L.N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P.: Combined iterative and model-driven optimization in an automatic parallelization framework. In: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE (2010)
20. Qasem, A., Kennedy, K.: Profitable loop fusion and tiling using model-driven empirical search. In: Proceedings of the 20th annual international conference on Supercomputing. pp. 249–258 (2006)
21. Rahman, M., Pouchet, L.N., Sadayappan, P.: Neural network assisted tile size selection. In: International Workshop on Automatic Performance Tuning (IWAPT'2010). Berkeley, CA: Springer Verlag (2010)
22. Rivera, G., Tseng, C.W.: A comparison of compiler tiling algorithms. In: International Conference on Compiler Construction. pp. 168–182. Springer (1999)
23. Shirako, J., Sharma, K., Fauzia, N., Pouchet, L.N., Ramanujam, J., Sadayappan, P., Sarkar, V.: Analytical bounds for optimal tile size selection. In: International Conference on Compiler Construction. pp. 101–121. Springer (2012)
24. Teixeira, S.T., Ancourt, C., Padua, D., Gropp, W.: Locus: a system and a language for program optimization. In: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 217–228. IEEE (2019)

25. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. pp. 1–12. IEEE (2009)
26. Wright, R.E.: Logistic regression. (1995)
27. Yang, Y.Q., Ancourt, C., Irigoin, F.: Minimal data dependence abstractions for loop transformations. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 201–216. Springer (1994)
28. Yotov, K., Li, X., Ren, G., Garzaran, M., Padua, D., Pingali, K., Stodghill, P.: Is search really necessary to generate high-performance blas? Proceedings of the IEEE **93**(2), 358–386 (2005)
29. Yuki, T., Renganarayanan, L., Rajopadhye, S., Anderson, C., Eichenberger, A.E., O'Brien, K.: Automatic creation of tile size selection models. In: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization. pp. 190–199 (2010)