COLA-Gen: Active Learning Techniques for Automatic Code Generation of Benchmarks

Maksim Berezov
Mines Paris, PSL University, France

Corinne Ancourt
Mines Paris, PSL University, France

Justyna Zawalska
Mines Paris, PSL University, France

Maryna Savchenko
Mines Paris, PSL University, France

Abstract

Benchmarking is crucial in code optimization. It is required to have a set of programs that we consider representative to validate optimization techniques or evaluate predictive performance models. However, there is a shortage of available benchmarks for code optimization, more pronounced when using machine learning techniques. The problem lies in the number of programs for testing because these techniques are sensitive to the quality and quantity of data used for training.

Our work aims to address these limitations. We present a methodology to efficiently generate benchmarks for the code optimization domain. It includes an automatic code generator, an associated DSL handling, the high-level specification of the desired code, and a smart strategy for extending the benchmark as needed.

The strategy is based on Active Learning techniques and helps to generate the most representative data for our benchmark. We observed that Machine Learning models trained on our benchmark produce better quality predictions and converge faster. The optimization based on the Active Learning method achieved up to 15% more speed-up than the passive learning method using the same amount of data.

2012 ACM Subject Classification Software and its engineering → Source code generation; Computing methodologies → Active learning settings

Keywords and phrases Benchmarking, Code Optimization, Active Learning, DSL, Synthetic code generation, Machine Learning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.3

Supplementary Material *Software (Source Code)*: `https://github.com/cri-internship/loop-generator`

Acknowledgements We want to thank Patryk Kiepas for productive discussion and ideas that helped this research to be finished.

1 Introduction

Benchmarking is an essential part of testing code optimization techniques and models. Such benchmark programs should be representative and reflect similar code characteristics that we are targeting.

Our objective is to have benchmarks adapted to the evaluation of source-to-source code transformations. These transformations make it possible to improve program characteristics such as the spatial and temporal locality of the data accesses, the loop iteration order, the potential parallelism. The execution time gain of the transformation depends on the transformation parameters that have to be instantiated for each kernel and the target architecture.

There are various known benchmarks for the C programming language that address specific aspects. For instance, BEEBS Benchmarks [18], Embench™ [2], MiBench [10] address different aspects of performance on embedded platforms. PolyBench 4.2 [23], Livermore loops (LFK) [17], LCALS v1.0.2, TSVC, [16], LORE [3] focus mainly on compiler optimizations and performance analysis. However, these benchmarks contain a limited number of typical kernels. For instance, TSVC contains 151 perfectly-nested loops, PolyBench 4.2 contains 30 computational kernels (kernel may contain several loop nests), Livermore loops (LFK) have 30 loop-nests, and LCALS v1.0.2 contains 32 loop-nests, LORE aggregates loops from other benchmarks and contains 2499 loops in C.

This amount of data may not be enough when code optimization actively uses Machine Learning (ML) techniques. The strength of ML techniques often comes from the use of a large training set. For instance, MNIST [22] a benchmark for image processing contains 70,000 images, LibriSpeech [19] for speech recognition includes 1000 hours of speech, Enron corpus [12] for natural language processing aggregates 500,000 messages.

Therefore, there are much less data in the code optimization domain than in the fields where ML is running at its peak performance. There is not enough training data to properly cover the feature space for complex transformations such as loop tiling, loop unrolling, loop interchange, etc. Also note that different transformations have different feature spaces from the ML perspective. One training set may capture better features for one transformation, another set – for a different transformation. It becomes challenging or even impossible to create a universal training set.

There are two main approaches to solve this problem: data mining [11, 8, 9] and synthetic code generation [7, 6, 4]. Nevertheless, data mining approaches have drawbacks such as data accuracy, completeness, parsing difficulties, libraries they may depend on, etc. In our study, we investigate the approach of synthetic code generation. Existing approaches either rely on the known predefined statistical distribution of the parameters [4] or require a huge training set for the deep learning model to mimic the given distribution [6].

Our work proposes a solution to these problems. We introduce a methodology to generate a representative benchmark that captures many computation patterns crucial for parallel computations. We let the ML model decide which data to include in the training benchmark among many potential candidates to achieve the best result. These candidates were not even compiled. Also, we present a code generator which can automatically create synthetic data. Our code generator uses information like array sizes, data-dependencies, loop index order, and data access functions as a high-level specification of the generated code. We use a DSL to easily manipulate these concepts and generate code in a very parametric and flexible way. Active learning methods allow us to direct code generation to the target function of the ML model. This data generation approach enables the creation of representative training sets for program optimization in the ML context. Moreover, we are able to generate our codes for different benchmark distribution styles, such as PolyBench.

This paper is structured as follows. Section 2 presents the context of our work and pointing out the guidelines we used in our code generator. Section 3 introduces our automatic code generator of C kernels. Section 4 presents the ML pipeline we use to get the predictions of the transformation parameters in the context of code optimization. Finally, Section 5 presents the data augmentation process with active learning techniques and its promising experimental results.

2 Context

To apply code optimizations, some transformation parameters must be defined. For example, to apply loop unrolling we have to predict the number of iterations to unroll, to apply loop tiling, the block sizes have to be selected. We follow three concepts as guidelines for building an automatic code generator of programs, used by ML techniques to predict efficient transformation parameters.

First, ML algorithms build prediction models based on training data. The key idea is that the model should capture meaningful patterns in training data and should be able to generalize them for arbitrary input. We use such high-level concepts for the code generator, which make it easy to mimic different code distributions and extend them as needed. We describe this strategy in Sections 3.

The second important aspect is the amount of available data. From the ML perspective, the more data available, the better, the more insights we can obtain. However, data labeling can be a very time-consuming process. Therefore, time also constraints the size of the training set. It is important to build a representative benchmark of a reasonable size.

We solve this problem by using active learning methods. The main idea of Active Learning is that not all points in a training set have the same impact on the model training and its final performance. The goal is to select only the most representative samples from the training set in order to match the time constraints. This issue is discussed in detail in Section 5. We compare this approach with classical passive learning techniques, where all points from the training set are treated equally and have the same probability of being taken to the final training set. In contrast, active learning methods assign a score to each point that corresponds to the profit that we can gain if we take this point for training.

The third aspect concerns the code characteristics. Because loop nests are often the time-consuming computation parts in programs, our study focuses on optimizations commonly used by compilers (e.g. loop permutation, tiling) that could potentially exploit all the benefits of parallel execution. To optimize their execution time, it is necessary to take into account the spatial and temporal locality of the data accesses and data dependencies to extract the potential parallelism and apply the transformations only when they are legal.

In this paper, we evaluate our methodology on the tiling transformation. 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. We consider 3-D cubic tiling, which means that we split 3-D iteration space by cubic tiles. The goal of Machine Learning is to predict the sizes of the tiles for each code. We investigate three feature spaces to address this problem: a) Yuki and al.[24], b) Liu and al.[15] and c) one-hot encoding of all array accesses.

We show that our methodology can accelerate the learning process in the context of given feature spaces by generating the most representative data.

3 Code Generator Design

In this section, we introduce the main components of our automatic generator of C code. For each of them, we specify the type of code generated. Figure 1 highlights its main building blocks.

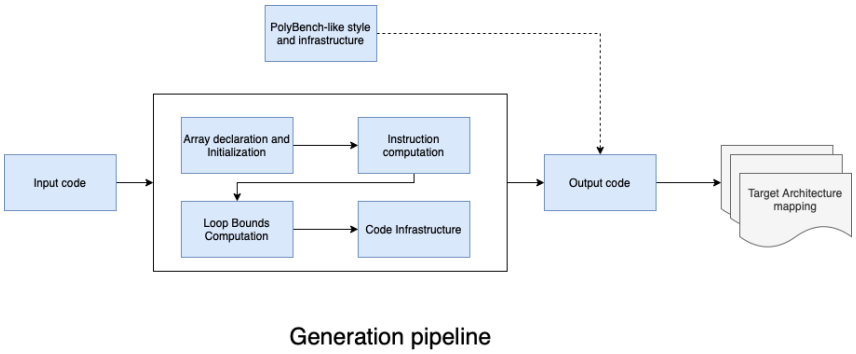

Figure 1 Generation pipeline.

3.1 Output Code and Input Data

The objective of the generator is to automatically produce a code written in C that respects the following hypothesis:

- it is correct. The code must not produce any runtime errors such as out of bounds memory access, etc;
- it meets the code criteria specified by the user in the DSL sample;
- it includes the necessary infrastructure to perform performance tests such as header files, *directives/pragmas* and calls to timing reporting functions;
- it can be compiled and executed. For instance, the arrays are properly initialized in the code.

In addition to these requirements, we use high-level input criteria for code optimization through a DSL. These are the number of arrays and their sizes (memory pressure), data dependencies, an order of the loop indices, and array access functions (pressure on spatial and temporal locality). These concepts allow us to explore the legality of potential transformations and optimize the code.

We apply the workflow of Figure 1 after parsing the input. After these steps, we get the first version of our code. Then there is the option to process the generated code to PolyBench-like style or another benchmark distribution style. The input example is shown in the appendix A, the corresponding output computations are shown in appendix B and the full code infrastructure is presented in appendix C.

3.2 Array Declaration and Initialization

The code generator takes array sizes from the input file (DSL description) and dynamically or statically (depending on the chosen option) allocates the requested arrays. The code generator may choose array sizes automatically if the user uses the PolyBench-like style of kernels. For instance, the EXTRALARGE_DATASET directive indicates that arrays should not fit the L3 cache.

3.3 Computation Instructions

This component generates the computation instructions included in the loop nest. Each instruction is composed of a reference to a write array and several (at least one) to read arrays. The array access functions are either explicitly given by the user or defined by the generator that respects the data dependencies which have been expressed in the DSL sample.

3.4 Loop Bound Computation

The generated code should be correct. To avoid out-of-bounds memory accesses to array elements, the generator computes the largest computation iteration domain according to the array declarations and the array access functions. We use linear-programming techniques to compute correct bounds. For constant dependencies, we generate numerical values for loop bounds.

3.5 Code Infrastructure

This component consists of adding all the infrastructure necessary for the execution of a stand-alone C program with time reporting functions. It includes: header files, variable declaration, and initialization, array allocation and deallocation, calls to time reporting functions, pragmas and directive insertion, and adjustment of array sizes according to the requested cache size, etc. We propose a processing pass that transfers our generated kernel to a PolyBench-like style.

4 Machine Learning modelling

Our objective is to show that our approach can accelerate the techniques of code optimization using ML in the context of a given feature space. In this section, we describe the pipeline that we would like to accelerate using Active Learning techniques. By accelerating, we mean the need for less training data to achieve good performance.

4.1 Machine Learning pipeline

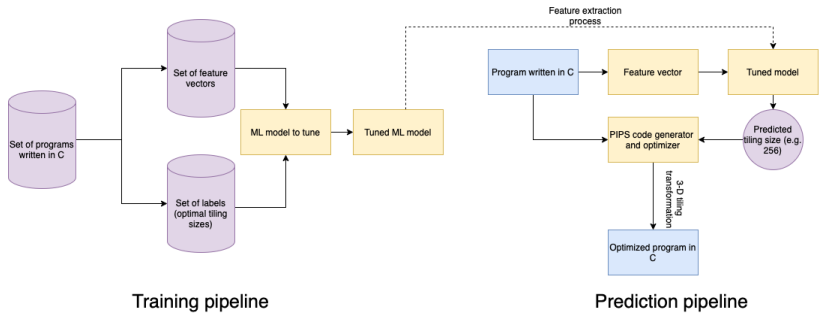

Figure 2 Training and prediction pipeline.

We investigate the problem of loop tiling size prediction for 3-D cubic tiles to validate our Machine Learning model. We consider tile sizes from 2 till 512 for the experiments and predictions. As features, we take the code characteristics proposed by a) Yuku et al. [24], b) Liu et al. [15] and c) one-hot encoding of array references.

We consider this problem as a regression problem. The model takes the features mentioned above as input and predicts the values of the tile sizes in the real domain. A heuristic of rounding the tile size to the nearest divisor of the loop bound could be applied and was used in our experiments. Then we generate the code based on predicted tile size. The training and prediction pipelines are shown in Figure 2.

Note that once the training pipeline phase is complete, the parameters of the prediction model are fixed. It is possible to predict with this tuned model the best parameters of the program transformation we want to apply in one shot. This model can then be integrated into a compiler.

A program Autotuner, such as LOCUS [20], typically uses several techniques to traverse a solution space and find an optimal version of a program. But the time needed to reach this solution for a program is not comparable to that of a single one-shot prediction of a tuned machine learning model. For this reason, we use the result of the Autotuner only as a reference of the optimal version to compare with our best predicted version of the program.

4.2 Machine Learning models

Non-linear machine learning models are more appropriate for our problem. To illustrate this point, we consider the case where the loop nest to be optimized is potentially vectorizable. There exist cases where the model must solve the following dilemma: If the loops are parallel, then tiling the innermost loop is the best, but this may be contrary to the optimization strategy of maximizing data locality. It can be seen as a decision tree. This is the main prerequisite for using nonlinear models for this task.

Random Forest regressor [14] showed the best results in terms of metrics considered in our experiments. This model was used to plot all the predictions of this paper.

4.3 Metrics

The mean squared error (MSE) loss was used as a Loss Function for regression.

$MSE = \frac{1}{n}\sum_{t=1}^{n}(y_i^* - y_i)^2$, where y_i is the ground-truth value of the optimal tile size of the i-th data sample, and y_i^* was predicted by our ML model. We use this metric for ML modelling since optimal tile sizes are distributed near the same neighborhood, and we want to penalize our model if it predicts tile sizes that are far from the global optimum.

This cost function has several drawbacks. It does not provide explicit information about our target goal - fast code execution. The loss provides no information to the programmer on how the generated code would perform in terms of execution time. Moreover, it does not provide insights about architecture parallelism and the profitability that we can gain from the transformation.

That is why we introduce the second-step metric showing how far we are from the most efficient generated code. We use the following relative speedup metric.

$RS_i = \frac{speedup(\hat{y}_i)}{speedup(y*_i)}$, where $speedup(\hat{y}_i)$ gives the speedup obtained after tiling the code with the predicted parameter. And $speedup(y*_i)$ gives the speedup found by the Autotuner. An average relative speedup can be computed with $RS = \frac{1}{n}\sum_{i=1}^{n} RS_i$.

The drawbacks of this function are that it is very sensitive to outliers. RS of a tile in the same neighborhood could be different due to factors that are not possible to take into account using existing feature spaces. Moreover, it does not have derivatives; it is a piecewise-defined function. Hence, it is not applicable to be used for training of many ML models. Thus, each metric is more appropriate for the stage where it is used. The combination of both provides a more correct way to navigate the training process and evaluate the results.

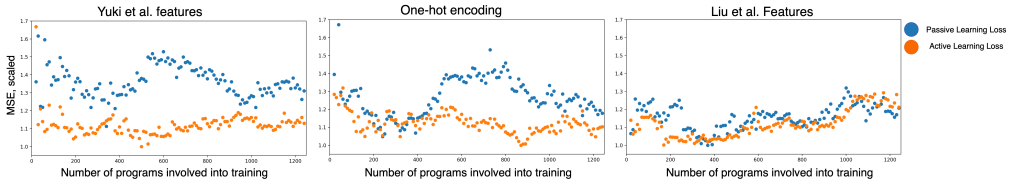

Figure 3 MSE on validation set.

5 Active Learning

Data labeling is the calculation of a value of the target variable for a data sample of the training set. This step can be very time-consuming in traditional ML pipelines. It makes sense to find a trade-off between how quickly we collect data and the accuracy of the final model. The issue of optimal experimental design arises. How to construct our training set to get the maximal possible gain? The techniques used in the active learning domain seem a promising direction to answer this question.

5.1 Active Learning Overview

Active Learning is a sub-field of Machine Learning. The crucial idea is that the model itself decides which data to use for more effective training. It finds thought in areas where data annotation is relatively expensive or may be not feasible.

Active Learning pipelines work under several scenarios: pool-based scenario [13], stream-based selective sampling [5] and membership query synthesis [1] The pool-based approach seems the most suitable for code optimization because it is the richest for a relatively low-cost. This approach assumes that we have a relatively small pool of annotated data and a much larger pool of not annotated data. At each iteration of the pool-based approach, the algorithm ranks all samples from the big pool according to a function. This function is chosen so that it returns a high value to the samples that have the potential to increase the performance of the ML model. The algorithm sends a query to the annotator to get labels for these samples. Then these annotated samples are added to the small pool of annotated data.

Sampling Strategies. The sampling strategy generates a query to the annotator in a pool-based scenario. In this subsection, we introduce the sampling strategies that can be used for supervised learning. In our experiments, we studied three approaches proposed by Wu et al.[21] for a regression problem.

- Greedy Sampling on the Inputs. The main idea is to choose the initial point as the closest to the centroid of the global pool, and then iteratively choose points farthest from the one already chosen to increase the diversity of the data in a given feature space.
- Greedy Sampling on the Outputs. The key idea is to use greedy sampling on the inputs to build the initial model, then to choose points with the farthest distance but in the output space according to the model prediction.
- Improved Greedy Sampling on both Inputs and Output. This approach considers the multiplication of the distances in the input and output spaces as the deciding metric. The data sample with the highest value is chosen.

5.2 Experimental statement

The learning process goes more efficiently for data generated with active learning, especially when we do not have expert knowledge about the given domain. We expose this statement to demonstrate the applicability of active learning techniques for the code optimization domain. While any handwritten strategy brings some bias to data, especially in case the expert knows which benchmarks will be used for testing, active learning appears to be the approach to facilitate representative data generation without introducing significant bias.

The pipeline for training the model is shown in Figure 2. The set of C programs could be obtained using naive sampling (passive learning) or more sophisticated strategies (active learning). The quality of the predictions and the speed of convergence of the models depends on this set.

5.3 Generating strategy

Training, test, and validation sets are required to properly tune the model and evaluate its applicability for real problems.

We train the ML model on the training set. The validation set is needed to evaluate the model performance (MSE) and determine its parameters based on that. Test set represents real-world data. We use our generator to sample data for the training and validations sets. We use a simple generation strategy that does not require any expert knowledge about the feature space for the loop tiling size prediction. The most important parameters that we vary are: existence of data dependencies, number of statements and array involved into the computations, loop index permutations.

Ten thousand kernels were randomly sampled to obtain a pool of not annotated data. Then, the Active Learning phase chooses 1250 most suitable kernels (training set for Active Learning). We do the labeling of chosen samples and train the model on them. Three hundred kernels were sampled (from the same distribution as 10k kernels) and labeled for validation set. There are not involved into model training but used as intermediate evaluation of the performance.

Nine known computational kernels were taken to form our test set. We compute the average relative speedup for them after tiling to assess the quality of the generated code.

5.4 Passive Learning Training Set

We sample the same amount (1250) of kernels with a random sampling to compare the performance of the model trained on the training set obtained with Active Learning. These 1250 kernels were chosen randomly also from 10k samples of not labeled data.

We investigate the possibility of Active Learning to shift the distribution to meaningful patterns in a given distribution.

5.5 Data labelling

The data labeling process begins after the choice of the kernels of the training set. This process is very time-consuming. For each kernel, we generate about 300 code variants (tiled codes with different tile sizes) and execute them to assign labels for the regression problem. The time to propose a variant plus its execution time varies from 0.1s to 50s, the median value is about 2s.

The whole process is equal to number of repetitions $\times$ number of variants $\times$ number of kernels $\times$ (the time to generate a variant + the time to execute the variant). For us it took around 30 days to label all the required data. This estimation illustrates that the data labeling process time can be significant. When time is limited, data quality becomes crucial. This is the main motivation for using the Active Learning approach.

5.6 Experimental results

The objective in this paper is not to find the best ML algorithm to perform tiling but to propose efficient techniques to automatically generate benchmarks suitable for the evaluation of code transformations and used as input for the ML techniques. In this section, we compare the results obtained with the Active and Passive Learning approaches.

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 Threads: 4, Opt. level: -O3

5.6.1 Loss on the validation set

Figure 3 compares the MSE on the validation set for the Active Learning approach and the Passive Learning approach for 3 different feature spaces. MSE was scaled by the minimal found value for the Active Learning strategy for the corresponding feature space. At some point, the losses for both strategies converge. But Active Learning significantly overperforms (for Yuki et al. and One-hot encoding) Passive learning under current settings due to the choice of the most diverse data. This fact could be used for problems where we have time constraints for data labeling and we need a faster-converged ML model.

5.6.2 Losses on the test set

We measure the average relative speedup for the test set to evaluate the quality of the generated code. Figure 4 shows the results for nine well-known computational kernels after applying loop tiling and for three different feature spaces. The blue columns correspond to the training process based on passive learning settings, the red ones - based on Active Learning.

The other columns correspond to speedups obtained with the state-of-the-art LOCUS auto-tuner [20] when loop tiling is applied. The autotuner's search space is made up of the same points as for our ML model (integer values from 2 till 512 for 3-D cubic tiling). LOCUS was asked to execute 300 points out of the search grid to find its best solution. These last results are used as references to know how far we are from the optimum.

Figure 5 introduces the relative average speedups for the three different feature spaces. The average relative speedup with the Yuki et al. [24] features obtained by the Active Learning is 71% out of the speedup found by LOCUS autotuner. The average speedup obtained by the Passive Learning is 58%. The same result is observed for the one-hot encoded features. The average speedup with Active Learning is 69% compared to 64 % without it. The corresponding values for Liu et al. [15] features are 68% and 53%.

Active Learning performs better than passive learning on average and for the majority of kernels. The average speedup along feature spaces is 1.11x higher with the use of Active Learning. The results obtained show that the active learning approach can traverse the learning process more efficiently and shift the distribution of chosen kernels towards important patterns. For the results shown in this paper, we used the Greedy Sampling on both Inputs and Outputs since it achieved the best quality.

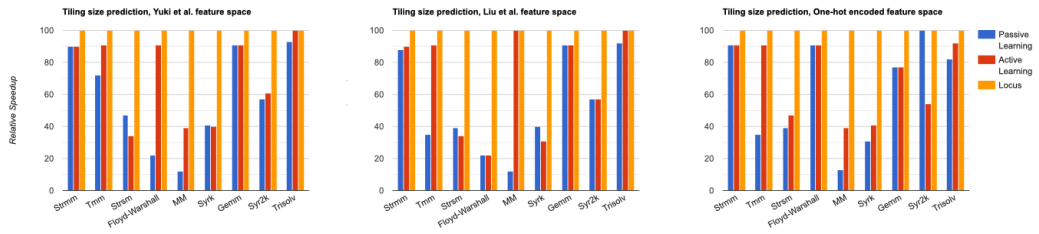

Figure 4 Average relative speedup for the test set.

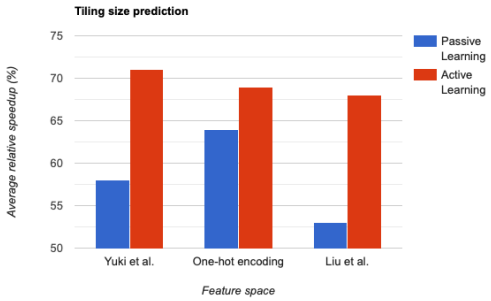

Figure 5 Average relative speedups.

6 Conclusion

This paper presents a methodology for efficiently generating benchmarks for code optimization using ML techniques. It includes 1) an automatic code generator enabling to imitate some existing benchmark styles 2) a smart strategy with active learning for extending the benchmark as needed.

We have proposed a strategy to increase the amount of data in a limited time. In this way, we only generate the most useful inputs. This approach allows us to select the best data for analysis and generate the most representative machine learning models if we do not have enough expert knowledge about the domain or do not want to introduce bias in the selection. The speedup gain for our strategy is up to 15% higher depending on the feature space and 11% higher on average.

Our future improvements targets extending the number of possible transformations and exploring more Active Learning techniques.

Our generator can be extended to many programming languages (not only C) because the main concepts we used are language-agnostic. It only requires a few modifications to the syntax and code routines to achieve a successful translation into the target language.

References

1 Dana Angluin. Queries revisited. In *International Conference on Algorithmic Learning Theory*, pages 12–31. Springer, 2001.

2 J Bennett, P Dabbelt, C Garlati, GS Madhusudan, T Mudge, and D Patterson. Embench: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative.

3 Zhi Chen, Zhangxiaowen Gong, Justin Josef Szaday, David C Wong, David Padua, Alexandru Nicolau, Alexander V Veidenbaum, Neftali Watkinson, Zehra Sura, Saeed Maleki, et al. Lore: A loop repository for the evaluation of compilers. In *Intern. Symp. on Workload Characterization (IISWC)*, pages 219–228. IEEE, 2017.

4 Alton Chiu, Joseph Garvey, and Tarek S Abdelrahman. Genesis: a language for generating synthetic training programs for machine learning. In *12th ACM Intern. Conf. on Computing Frontiers*, pages 1–8, 2015.

5 David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine learning*, 15(2):201–221, 1994.

6 Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *Intern. Symp. on Code Generation and Optimization (CGO)*, pages 86–99. IEEE, 2017.

7 Etem Deniz and Alper Sen. Minime-gpu: Multicore benchmark synthesizer for gpus. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):1–25, 2015.

8 Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *24th ACM SIGSOFT intern. Symp. on foundations of software engineering*, pages 254–265, 2016.

9 Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *39th Intern. Conf. on Software Engineering Companion (ICSE-C)*, pages 501–502. IEEE, 2017.

10 Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *4th Intern. workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

11 Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *11th working conf. on mining software repositories*, pages 92–101, 2014.

12 Bryan Klimt and Yiming Yang. Introducing the enron corpus. In *CEAS*, 2004.

13 David D Lewis and William A Gale. A sequential algorithm for training text classifiers. In *SIGIR'94*, pages 3–12. Springer, 1994.

14 Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

15 Song Liu, Yuanzhen Cui, Qing Jiang, Qian Wang, and Weiguo Wu. An efficient tile size selection model based on machine learning. *Journal of Parallel and Distributed Computing*, 121:27–41, 2018.

16 Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. An evaluation of vectorizing compilers. In *Intern. Conf. on Parallel Architectures and Compilation Techniques*, pages 372–382. IEEE, 2011.

17 FH McMahon. Livermore fortran kernels: A computer test of numerical performance range ucrl-53745. *LLNL, CA., USA*, 1986.

18 James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint*, 2013. `arXiv:1308.5174`.

19 Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *Intern. conf. on acoustics, speech and signal processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

20 SFX Thiago Teixeira, Corinne Ancourt, David Padua, and William Gropp. Locus: a system and a language for program optimization. In *Intern. Symp. on Code Generation and Optimization (CGO)*, pages 217–228. IEEE, 2019.

21 Dongrui Wu, Chin-Teng Lin, and Jian Huang. Active learning for regression using greedy sampling. *Information Sciences*, 474:90–105, 2019.

22 Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*, 2017. `arXiv:1708.07747`.

23 T. Yuki and L. Pouchet. Polybench 4.2, Jan 26, 2021. URL: `https://sourceforge.net/projects/polybench/`.

24 Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *8th Intern. Symp. on Code Generation and Optimization*, pages 190–199, 2010.

A Generated code examples. Input file.

Listing 1 JSON-like DSL specification.

```
[{"array_sizes": {"xA": 64, "yA": 32, "zA": 128}, "type": "int",
"init_with": "random", "loop_nest_level": 3,
    "arrays": ["A[xA,yA,zA]", "B[256,256]"],
    "instructions": [{"array_name": "A",
        "index_permutation": "(1,0,2)",
        "dependencies": {"distance": "[(1,2,3)]"},
        "additional_computation": [{"array_name": "B",
            "array_access_function": "[[0,2,0,8], [1,1,1,8]]"}]}]}]
```

B Generated code examples. Output computations.

Listing 2 Generated computations.

```
int A[64][32][128],B[256][256];
  ....
for (int i = 0; i < 30; i++)
for (int j = 0; j < 63; j++)
for (int k = max(-i-j-8, 0); k < min(248-i-j, 125); k++)
    A[j][i][k]=A[j+1][i+2][k+3]+B[2*j+8][i+j+k+8];
```

C Generated code examples. Full code infrastructure.

Listing 3 PolyBench style generated code.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <math.h>
#include <polybench.h>
#include <stdio.h>
# include <time.h>
# include <stdlib.h>
#include "1648808249866439.h"
static void init_array(int xa,int ya,int za,
DATA_TYPE POLYBENCH_3D(A,xA,yA,zA,xa,ya,za),
int xb,int yb,DATA_TYPE POLYBENCH_2D(B,xB,yB,xb,yb))
{ srand(time(NULL));
int i,j,k,l;
for (i = 0; i < xa; i++)
    for (j = 0; j < ya; j++)
        for (k = 0; k < za; k++)
            A[i][j][k] = rand()%50;

for (i = 0; i < xb; i++)
    for (j = 0; j < yb; j++)
        B[i][j] = rand()%50;
}
```

```
static void print_array(int xa,int ya,int za,
DATA_TYPE POLYBENCH_3D(A,xA,yA,zA,xa,ya,za),
int xb,int yb,DATA_TYPE POLYBENCH_2D(B,xB,yB,xb,yb))
{ int i,j,k,l;
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("A");
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("A");
for (i = 0; i < xa; i++) {
for (j = 0; j < ya; j++) {
for (k = 0; k < za; k++) {
fprintf (POLYBENCH_DUMP_TARGET, "\n");
fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, A[i][j][k]);
}}}
POLYBENCH_DUMP_END("A");
POLYBENCH_DUMP_FINISH;
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("B");
POLYBENCH_DUMP_START;
POLYBENCH_DUMP_BEGIN("B");
for (i = 0; i < xb; i++) {
for (j = 0; j < yb; j++) {
fprintf (POLYBENCH_DUMP_TARGET, "\n");
fprintf (POLYBENCH_DUMP_TARGET, DATA_PRINTF_MODIFIER, B[i][j]);
}}
POLYBENCH_DUMP_END("B");
POLYBENCH_DUMP_FINISH;
}
int main(int argc, char** argv)
{
int xa = xA;
int ya = yA;
int za = zA;
int xb = xB;
int yb = yB;
POLYBENCH_3D_ARRAY_DECL(A, DATA_TYPE,xA,yA,zA,xa,ya,za);
POLYBENCH_2D_ARRAY_DECL(B, DATA_TYPE,xB,yB,xb,yb);
init_array(xa,ya,za,POLYBENCH_ARRAY(A),xb,yb,POLYBENCH_ARRAY(B));
kernel_1648808249866439(xa,ya,za,POLYBENCH_ARRAY(A),xb,yb,
POLYBENCH_ARRAY(B));
polybench_prevent_dce(print_array(xa,ya,za,
POLYBENCH_ARRAY(A),xb,yb,POLYBENCH_ARRAY(B)));
POLYBENCH_FREE_ARRAY(A);
POLYBENCH_FREE_ARRAY(B);
return 0;
}
void kernel_1648808249866439(int xa,int ya,int za,
DATA_TYPE POLYBENCH_3D(A,xA,yA,zA,xa,ya,za),int xb,int yb,
DATA_TYPE POLYBENCH_2D(B,xB,yB,xb,yb)){
polybench_start_instruments;
#pragma scop
tiling_3D: for (int i = 0; i < 30; i++)
tiling_2D: for (int j = 0; j < 63; j++)
for (int k = max(-i-j-8,0); k < min(248-i-j,125); k++)
               A[j][i][k]=A[j+1][i+2][k+3]-B[2*j+8][i+j+k+8];
```

```
clock_t stop = clock();
double elapsed = ((double)(stop - start)) / CLOCKS_PER_SEC;
printf("%f", elapsed);
deallocate_3d_array(A, 64, 32, 128);
deallocate_2d_array(B, 256, 256);
return 0;
}
```