End-to-end Deep Learning of Compiler Optimization Heuristics (Group 16)

Cheng-Yu Lin, Wen-Chien Wang, Chieh-Hao Chuang, Chun-Hsiang Chan
University of Michigan, Ann Arbor, MI, USA
{chenyul, wcwang, chchuang, kenhchan}@umich.edu

Abstract

We face plenty of decision-making scenario on choosing optimization methods, during the compilation and execution. Modern compilers resolve this issue by extensive hand-coded heuristics. In contrast, researchers utilize machine learning techniques to build more flexible heuristics recently. We reproduce the deep learning framework introduced by Leather, et al. [8], and utilize this architecture to solve two problems: (a) mapping source code to the optimal device (CPU/GPU) and (b) predicting the thread coarsening factor. In addition, we improve performance by training the model by the hybrid vectors of source code and runtime data.

1. Introduction

To optimize the runtime of a code, the compiler needs to choose the optimization plan. That is, the compiler needs to decide a particular optimization should be applied or not on executing the program. We reproduce a compiler decision heuristic, based on the work of Leather, et al. [8] and demonstrate such deep-learning architecture can serve for (a) building a predictive model for mapping OpenCL kernels to the optimal device in CPU/GPU heterogeneous systems; (b) building a decision heuristic on predicting the OpenCL thread coarsening factor. In addition, we improve the accuracy of such a deep-learning model on task (a) by training the model with not only the characteristic vector (automatically generating by the feature extraction networks with source code input [8]) but also the features selecting from the runtime data.

Traditionally, the compiler utilizes hand-written heuristics along with the profiling data to decide the optimization strategies. However, hand-written heuristics have several drawbacks. For example, hand-written heuristics require expert knowledge, and plenty of time to construct distinct heuristics for different optimization tasks. Besides, the combination of several distinct heuristics often leads to sub-optimal decisions in a global sense. Instead, researchers utilize machine learning techniques to build more flexible heuristics, which outperform the hand-written heuristics from several perspectives [22, 12, 24, 11, 5, 10, 2]. Those models use supervised machine learning, on runtime (or profiling) data and learn the correlation between the selected features and the optimization decision. Though such models outperform traditional hand-written heuristics, they still have several downsides that can be improved. For instance, those heuristics still based on runtime (or profiling), and this could increase the runtime overhead during the compiling. Besides, feature selection is critical in training the models. Misidentifying an important feature will dramatically decrease the heuristic performance. For example, authors in [8] discovered a missing feature, that can increase the performance of the model in [6] by over 40%.

Leather, et al. [8] addresses both of the issues by a deep-learning architecture. They provide a deep-learning network to extra the feature from the source code and vectorize them automatically. The high light is that this DNN based feature extraction technique does not rely on profiling (or runtime data) and they can turn the source code directly into a set of trainable data. In the feature extraction framework, they first utilize an LLVM pass to rebuild the input source code into a consistent style code. Next, they use the tokenized approach [3] to parse the code. With a stander natural language processing technique, each parse will be the map to a point in the vector space. Finally, each source code will be turned into a set of vectors and using the LSTM to embedding those data to a single characteristic vector. Intuitively, the characteristic vector can represent the relationship between different codes. That is, source codes with a similar function will be mapped to the nearby vectors in the vector space. They then use such characteristic vectors to train the classification models and sever for the decision-making task on several compiler optimization problems.

Besides, Leather, et al. [8] show that the feature extraction networks can be applied to different optimization problems with the same network parameters, by using transfer learning [17]. That is, the feature extracting networks for the source code is mostly independent of the optimization problem. Thus, by reusing such networks across different optimization problems, they can decrease the learning time dramatically.

We reproduce and modify the deep-learning architecture of Leather, et al. [8], and evaluated such architecture with two kinds of compiler optimization tasks. Overall, we make the following effort:

1. We reproduce and modify the end-to-end DNN model of Leather, et al. [8];

2. We examine the DNN model on (a) mapping source code to the optimal device (CPU/GPU) and (b) predicting the thread coarsening factor.

3. Case (a): We improve the model performance by nearly 10% by training the model with a concatenating vector of the characteristic vector and the runtime features. We note that the improvement is comparing to the models training only by either the characteristic vector or the runtime features.

4. Case (b): We reproduce the predicting heuristic of thread coarsening factor with the same DNN model, based on the transfer learning.

For case (a), it is important to have good heuristics for determining the execution device on heterogeneous systems. For case (b), thread coarsening is critical in applying GPU architectures.

We organize our work as follows. Section 2 is the related works. In section 3, we introduce the detailed constructions of the reproduced and modified models. In section 4, we introduce the experimental setting of the case (a) and (b). In section 3, we describe the experimental results and we give a comprehensive analysis as well as a comparison for the experiments. Finally, we end up our works with the conclusion in section 6.

2. Related Work

Machine learning techniques have been applied to model and learn from program source code on several different tasks, such as mining coding conventions [19] and idioms [4], API example code [27], and pseudo-code generation [28], and benchmark generation [7]. On the other hand, machine-learning has also intensively engaged in compiler optimization [26, 18, 23, 25, 16, 6]. The research used to train the model by selecting features from profiling data, static code structures, runtime information, and performance counters [9]. For example, Cavazos et al. profile the target program using several carefully selected compiler options to see how to program runtime changes under these options [15]. However, these kinds of approaches have several drawbacks. First, those models are still based on profiling and runtime data. That is, the program must be run several times before optimization. Second, developers must carefully select a few features from a large number of candidate options, because poorly chosen options can significantly affect the quality of the model.

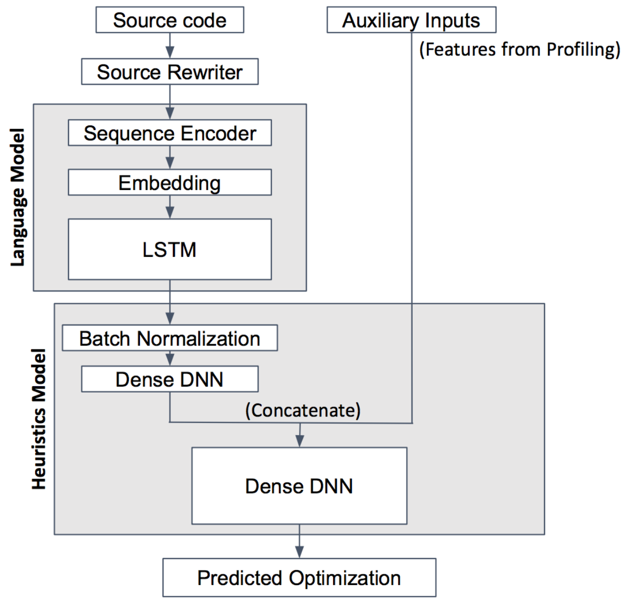

Figure 1. Code properties are extracted from source code by the language model(feature extraction networks). The language model will output a 2048 dimensions vector. After batch normalization and a fully-connected layer, we compress the 2048 dimensions vector into a 32 dimensions vector. We then concatenate such vector with a 6 dimensions auxiliary inputs (6 features from runtime data), and the concatenated vector is fed into the two fully-connected layers to produce the final prediction.

Recently, several researchers try to automize the feature extraction task while training the model on compiler optimization problems. A few methods have been proposed to automatically generate features from the compiler's intermediate representation [21, 13]. The work of [13] uses genetic programming to search for features and the work of [21] expresses the space of features via logic programming. However, both works require significant search time.

Leather, et al. [8] try to attack the compiler optimization problem more ambitiously. They train the model to address several compiler optimization problems by not only extract the meaningful feature automatically but also extract the features from the source code structure without profiling or runtime information. Their work is the first study on deep neural networks for program feature generation without additional runtime or profiling data.

3. Methods

Based on the work of Leather, et, al. [8], we reproduce the DNN architecture with some modifications to improve the performance. Figure 1 provides an overview of the system. There are two main building blocks in our system. One is the language model, and the other is the heuristic model. The language model can extract the feature directly from the source code and output the characteristic vectors. The heuristic model serves as the classification heuristic of the optimization problem. The system overview is as follow.

First, we use the source rewriter to formalize the source code (removing the irrelevant code, unifying the coding style) and then pass the formalize code to the language model. After passing the language model, we further combine the characteristic vectors with the feature selected from the runtime data to train our heuristic model. Remark that we utilize different concatenation strategy to combine the characteristic vectors and the runtime features. That is, we do not directly concatenate the characteristic vectors with the selected features. Instead, after the source code is turned into 2048 dimensions characteristic vectors, we reduce the characteristic vectors to 32 dimensions and then apply the concatenation. We implement this way because we only have six features selected from the runtime data. If we apply the concatenation directly, then the runtime features do not lead to a meaningful effect on the model. Thus, to increase the impact of that runtime feature, we first compress the characteristic vectors from 2048 to 32 dimensions.

In the rest of the section, we will introduce the details for the entire system.

3.1. Language Model

The language model includes four parts, the source rewriter, sequence encoder, embedding, and sequence characterization. To automatically extract the characteristic and generate trainable data from the given source code, the key idea is applying the natural language processing technique [7, 3] on the source code. That is, after formalizing the source code, Leather, et al. [8] use the tokenize technique to embedding the source code into a set of low-dimensional data points. Next, they use the Long Short-Term Memory (LSTM) architecture to turn such set of data into a 2048 dimensions characteristic vector. We detailly describe each part as follow:

Source Rewriter: The rewriter is implemented as an LLVM pass [7]. It contains the following function: (1) parse the AST; (2) removing conditional compilation, such as comments; (3) rewrite the input source code to a stander code style. This rewriting process enables us to roll out the irrelevant information of the source code, simplify the tokenize task, and let the language model extract the meaningful features easier.

Sequence Encoder and Embedding: The sequence encoder is a hybrid between the character-based model and the token-based model. This allows common keywords, such as "float", "int", and "if" to be represented as unique vocabulary items, while others are encoded at the character level. We encode the source code into a sequence of predetermined characters and vocabulary. The algorithm consumes the longest matching sequence from the candidate vocabulary. This process continues until every character in the corpus has been consumed. The resulting derived vocabulary consists of 128 symbols which we use to encode the source code. We then embed such tokenized sequence into a matrix space. Each token in the sequence is mapping to a vector of real values and the mapping ensures that the semantically related tokens like float and int to be mapped to nearby points [1, 20]. Formally, Given a vocabulary size V and embedding dimensionality D, an embedding matrix $W \in \mathbb{R}^{V \times D}$ is learned during training so that an integer encoded sequences of tokens $t \in \mathbb{N}^{L}$ is mapped to the matrix $T \in \mathbb{R}^{L \times D}$. We set D as 64.

Sequence Characterization: After source codes have been encoded into an embedding matrix, we use the Long Short-Term Memory (LSTM) architecture [14] to extract a 2048 dimension characteristic vectors from the entire sequence. The LSTMs are commonly used in feature extractions of the sequence of inputs. With a forget gate and a linear activation function, LSTMs possess a great ability to learn long ranges relations [29]. This property enables us to model the program code and extract the coding characteristic. We use a two-layer LSTM network. The network receives a sequence of embedding vectors, and returns a single output vector, characterizing the entire sequence. We call such output vector the characteristic vector throughout this article.

3.2. Auxiliary Inputs

Our architecture supports the augmentation between the source code input and the runtime input. We concatenate these two inputs in the heuristic model. In detail, after obtaining the 2048 dimension characteristic vector, we first compress it into 32 dimension vectors and then concatenate such vectors with the real-valued runtime feature. We reduce the dimensions by batch normalization and a single fully-connected layer. We provide this modification because real-valued auxiliary inputs usually have much fewer dimensions than the original characteristic vector. Thus, to balance the effect between two inputs, we apply the dimensions reduction technique there. Figure 2 show the six runtime feature we use in training the model.

3.3. Heuristic Model

Concatenated vectors are fed through the last two layers of the heuristic model to predict classification results. The last two layers are both fully-connected neural network layers. The first layer consists of 38 neurons. The second layer consists of a single neuron for each possible heuristic decision. For example, in mapping source code to the optimal device, two neurons are representing either CPU or GPU.

Name	Type	Description
`comp`	static	#. compute operations
`mem`	static	#. accesses to global memory
`localmem`	static	#. accesses to local memory
`coalesced`	static	#. coalesced memory accesses
`data size`	dynamic	size of data transfers
`workgroup size`	dynamic	#. work-items per kernel

Figure 2. We train our model with the source code data and the six runtime feature above. Note that Leather, et al. [8] only apply two of the runtime feature in training case (a).

Each neuron is an activation function unit that will trigger by the input. For the output layer, we use sigmoid activation functions and restrict the output in the range $[0, 1]$. We choose the decision with the largest output value, since the output of each neuron represents the model's confidence that the corresponding decision is the correct one.

4. Experimental Methodlogy

We use the reproduced architecture on two kinds of optimization tasks, and show some improvement in Case (a) with a modified network.

Case (a) OpenCL Heterogeneous Mapping: OpenCL allows the program to execute transparently across a range of different devices, from CPUs to GPUs. Thus, when we execute a program, we aim to know which device (CPUs, GPUs) can we maximize the performance of the given program? Grewe et al. [10] address this problem by constructing a decision tree via supervised learning. Their model is trained by both static and dynamic kernel features, where the static features get from a custom LLVM pass, and the dynamic features extract from the OpenCL runtime data. Leather et al. [8] provide an automatic feature extraction framework from the source code, and train the model by the source code, and the two features from the runtime data. Our architecture is similar to Leather et al. [8] with some modifications. First, we do not concatenate the source code characteristic vector with the runtime features directly. Instead, we compress the characteristic vector from 2048 to 32 dimensions and then concatenate it with runtime features. Second, we use six runtime features in the concatenations instead of two.

Experimental Setup: We use 192 different OpenCL kernel source code as our training data and 32 different OpenCL kernel source code as our testing data to evaluate each model performance. We set up four kinds of scenarios and evaluate the performance by the prediction accuracy. First, we take the ground truth as the correctness of randomly guessing the output label. Since there are 54% of the program having better performance in CPUs, the ground truth accuracy is 54%. Second, we use six runtime data to build a classification model on this problem. We implement the model through both logistic regression and support vector machine. Note that since there are only six runtime features, it is not much meaningful to apply a deep learning network in this setting. Third, we reproduce the architecture of Leather et, al.[], and train their model with the source code input only. Finally, we train our modified model by combining the characteristic vector of source code and the selected features from the runtime data.

Case (b) OpenCL Thread Coarsening Factor: The thread coarsening factor affects a reduction in parallelism in the GPU execution, which can have both a beneficial and detrimental effect on runtime. The coarsening factor decides the number of times the body of the thread is replicated and can lead to some advantage. For example, the coarsened code can reduce the number of threads to be launched in the program execution and the replication instructions provide the opportunity for hardware instruction-level parallelism. However, increasing the coarsening factor also raises resource consumption (i.e. registers) and will eventually result in decreasing performance. Thus, choosing a proper thread coarsening factor is crucial in parallel computation.

Thread coarsening as optimization has been extensively explored as a semi-automatic [2] and deep-learning-based [8] optimization. Magni et al. [2] present a predictive model for OpenCL thread coarsening. They implement an iterative heuristic which determines whether a given program would benefit from coarsening. If yes, then the program is coarsened, and the process repeats, allowing further coarsening. In this manner, the problem is reduced from a multi-label classification problem into a series of binary decisions. They select from one of six possible coarsening factors: (1, 2, 4, 8, 16, 32), divided into 5 binary choices. They followed a very comprehensive feature engineering process. Candidate features were assembled from the performance counters and computed theoretical values. On the other hand, Leather et, al [8] provide an auto feature extraction deep-learning model on this problem to take humans out of the loop. Their networks train only with source code without additional runtime or profiling information and apply transfer learning concepts on the feature extraction network.

Experimental Setup: We try to reproduce the results of Leather et, al [8]. They utilize transfer learning to reduce the need for training data. Magically, they get completive results with only 17 training data and validate the results by one of the data which random pick from the data set in each round. We also use the transfer learning technique in our reproduce model. We train our model same with the above setting and evaluate the performance by comparing it to the prediction results in Leather et, al [8].

5. Experimental Results

We show the experimental results of our model on two distinct optimization tasks: (a) predicting the optimal device to run a given program and (b) predicting thread coarsening factors.

Case (a) OpenCL Heterogeneous Mapping: Selecting the optimal execution device for OpenCL kernels is essential for maximizing performance. For a CPU/GPU heterogeneous system, this presents a binary choice. We train our model with 192 distinct source code and test the performance with 32 distinct sources. The accuracy is measured by the percentage of the model making a correct choice that optimizes the device runtime. Recall that we compare our modified approach with four kinds of scenarios and evaluate the performance by the accuracy. First, we set the baseline as randomly guessing the CPU/GPU label. The ground truth accuracy is 54% since there are 54% of the program having better performance in CPUs. Second, we build the binary decision model by six runtime features. We implement the model by the logistic regression and the support vector machine, and we get 60% accuracy and 62% accuracy, respectively. Third, we reproduce the architecture of Leather et, al. [8], and train their model with the source code input only. In this case, we get 59% accuracy. Finally, we train our modified model by combining the characteristic vector (32 dimensions after compression) and the six selected features from the runtime data and we improve the performance to 75% accuracy.

In either case (the model is trained only with the selected features from the runtime data or the model is trained only with the characteristic vectors extracting from the source code), we can not maximize the model performance. However, our experiment shows that properly combine the selected features from the runtime data and the characteristic vectors automatically generated from the source code can yield better performance. Figure 3 show the results in case (a).

Case (b) OpenCL Thread Coarsening Factor: We train our model with 17 distinct source codes and evaluate the performance with the results proposed by Leather et, al [8].

The baseline accuracy is randomly guessing the Thread Coarsening Factor. Since we have six possible values, the baseline accuracy is 16%. We then reproduce the model of Leather et, al [8] and train the model under 17 distinct source code data. We compute the accuracy by comparing our prediction results with the results provide in their paper. Unfortunately, our reproduce model only has 30% accuracy. Although we use the transfer learning technique on the language model, we utilize too small a data set. Thus, we get low accuracy on this task.

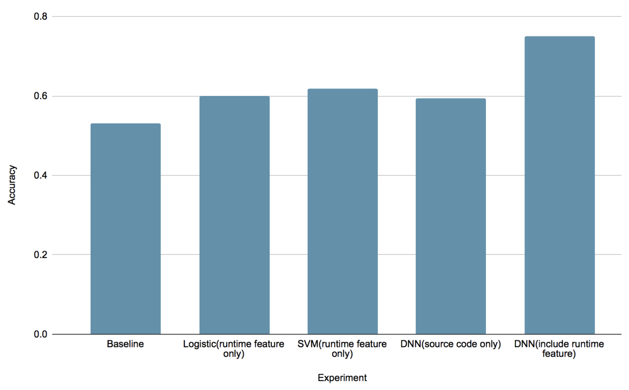

Figure 3. The model is train with 192 distinct source code and test with 32 distinct sources. the baseline is randomly guessing the CPU/GPU label. The logistic regression and the support vector machine model is trained by six runtime features. The architecture of Leather et, al. [8] is trained by the source code input only. The modified architecture is trained by both the source code input and the runtime data.

6. Conclusions

In this project, we reproduce the Leather et al. [8] deep-learning architecture for two compiler optimization problems. We examine the power of the auto feature extractor. That is, the model can extract the source code feature by tokenizing the code and embedding the information into characteristic vectors. This technique is powerful since it bypasses the profiling process, runtime analysis, and human-in-the-loop feature selection. In addition, we improve the reproduced model by concatenating the characteristic vectors with the selected runtime feature. This shows that properly selected runtime features are still valuable, and combining both of the information can increase the performance of the model significantly. However, we do not get satisfactory results in thread coarsening factors prediction. That is, we do not benefit from the transfer learning technique. To get better results, we should increase our data set.

Overall, this project enables us to look into an alternative techniques on the decision problem in compiler optimization and we further combine the traditional runtime data with the deep-learning approach to lead some improvement.

References

[1] Distributed representations of words and phrases and their compositionality.

[2] C. Dubach A. Magni and M. O'Boyle. Automatic optimization of thread-coarsening for graphics processors. *PACT. ACM*, 2014.

[3] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. *MSR*, 2013.

[4] M. Allamanis and C. Sutton. Mining idioms from source code. *FSE. ACM*, 2014.

[5] M. Steuwer C. Cummins, P. Petoumenos and H. Leather. Autotuning opencl workgroup size for stencil patterns. *ADAPT*, 2016.

[6] M. Steuwer C. Cummins, P. Petoumenos and H. Leather. Towards collaborative performance tuning of algorithmic skeletons. *HLPGPU*, 2016.

[7] W. Zang C. Cummins, P. Petoumenos and H. Leather. Synthesizing benchmarks for predictive modeling. *CGO. IEEE*, 2017.

[8] Z. Wang C. Cummins, P. Petoumenos and H Leather. End-to-end deep learning of optimization heuristics. *PACT. IEEE*, 2017.

[9] E. V. Bonilla G. Fursin C. Dubach, T. M. Jones and M. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. *MICRO. ACM*, 2009.

[10] Z. Wang D. Grewe and M. O'Boyle. Portable mapping of data parallel programs to opencl for heterogeneous systems. *CGO. IEEE*, 2013.

[11] J. Cavazos B. Franke G. Fursin M. O'Boyle J. Thomson M. Toussaint F. Agakov, E. Bonilla and C. K. I. Williams. Using machine learning to focus iterative optimization. *CGO. IEEE*, 2006.

[12] T. L. Falch and A. C. Elster. Machine learning based autotuning for enhanced opencl performance portability. *IPDPSW. IEEE*, 2015.

[13] E. Bonilla H. Leather and M. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. *TACO*, 2014.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 1997.

[15] F. Agakov E. Bonilla M. O'Boyle G. Fursin J. Cavazos, C. Dubach and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. *CASES*, 2006.

[16] L. Gao J. Ren and Z. Wang. Optimise web browsing on heterogeneous mobile platforms: A machine learning based approach. *INFOCOM*, 2017.

[17] Y. Bengio J. Yosinski, J. Clune and H. Lipson. How transferable are features in deep neural networks? *NIPS*, 2014.

[18] S. Kulkarni and J. Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. *OOPSLA. ACM*, 2012.

[19] C. Bird M. Allamanis, E. T. Barr and C. Sutton. Learning natural coding conventions. *FSE. ACM*, 2014.

[20] G. Dinu M. Baroni and G. Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. *ACL*, 2014.

[21] G. Fursin A. Zaks M. Namolaru, A. Cohen and A. Freund. Practical aggregation of semantical program properties for machine learning based optimization. *CASES*, 2010.

[22] A. Smith P. Micolet and C. Dubach. A machine learning approach to mapping streaming workloads to dynamic multicore processors. *LCTES. ACM*, 2016.

[23] M. Hall M. Garland S. Muralidharan, A. Roy and P. Rai. Architecture-adaptive code variant tuning. *ASPLOS. ACM*, 2016.

[24] M. Stephenson and S. Amarasingh. Predicting unroll factors using supervised classification. *CGO. IEEE*, 2005.

[25] Z. Wang W. F. Ogilvie, P. Petoumenos and H. Leather. Minimizing the cost of iterative compilation with active learning. *CGO*, 2017.

[26] Z. Wang and M. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based ap- proach. *PACT. ACM*, 2010.

[27] D. Zhang X. Gu, H. Zhang and S. Kim. Deep api learning. *FSE. ACM*, 2016.

[28] G. Neubig H. Hata S. Sakti T. Toda Y. Oda, H. Fudaba and S. Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (t). *ASE. IEEE*, 2015.

[29] J. Berkowitz Z. C. Lipton and C. Elkan. A critical review of recurrent neural networks for sequence learning. *arXiv:1506.00019*, 2015.