End-to-end Deep Learning of Optimization Heuristics

Abstract

Accurate automatic optimization heuristics are necessary for dealing with the complexity and diversity of modern hardware and software. Machine learning is a proven technique for learning such heuristics, but its success is bound by the quality of the features used. These features must be hand crafted by developers through a combination of expert domain knowledge and trial and error. This makes the quality of the final model directly dependent on the skill and available time of the system architect.

Our work introduces a better way for building heuristics. We develop a deep neural network that learns heuristics over raw code, entirely without using code features. The neural network simultaneously constructs appropriate representations of the code and learns how best to optimize, removing the need for manual feature creation. Further, we show that our neural nets can transfer learning from one optimization problem to another, improving the accuracy of new models, without the help of human experts.

We compare the effectiveness of our automatically generated heuristics against ones with features hand-picked by experts. We examine two challenging tasks: predicting optimal mapping for heterogeneous parallelism and GPU thread coarsening factors. In 89% of the cases, the quality of our fully automatic heuristics matches or surpasses that of state-of-the-art predictive models using hand-crafted features, providing on average 14% and 12% more performance with no human effort expended on designing features.

1. Introduction

There are countless scenarios during the compilation and execution of a parallel program where decisions must be made as to how, or if, a particular optimization should be applied. Modern compilers and runtimes are rife with hand coded *heuristics* which perform this decision making. The performance of parallel programs is thus dependent on the quality of these heuristics.

Hand-written heuristics require expert knowledge, take a lot of time to construct, and in many cases lead to suboptimal decisions. Researchers have focused on machine learning as a means to constructing high quality heuristics that often outperform their handcrafted equivalents [1–4]. A *predictive model* is trained, using supervised machine learning, on empirical performance data and important quantifiable properties, or *features*, of representative programs. The model learns the correlation between these feature values and the optimization decision that maximizes performance. The learned correlations are used to predict the best optimization decisions for new programs. Previous works in this area were able to build machine learning based heuristics that outperform ones created manually by experts and did so with less effort [5, 6].

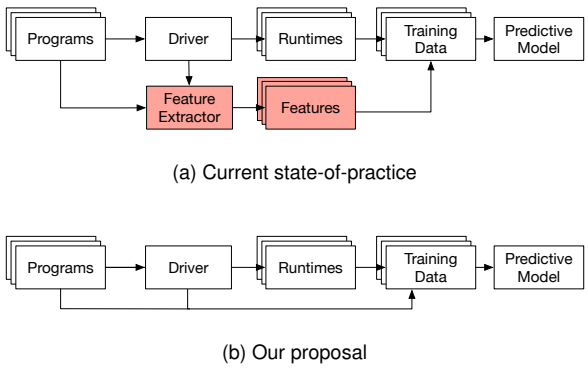


Figure 1: Building a predictive model. The model is originally trained on performance data and features extracted from the source code and the runtime behavior. We propose bypassing feature extraction, instead learning directly over raw program source code.

Still, experts are not completely removed from the design process, which is shown in Figure 1a. Selecting the appropriate features is a manual undertaking which requires a deep understanding of the system. The designer essentially decides which compile or runtime characteristics affect optimization decisions and expresses them in ways that make it easy to model their relationship to performance. Failing to identify an important feature has a negative effect on the resulting heuristic. For example, in [7] the authors discovered that [5] did not identify one such feature, causing performance to be 40% lower on average.

To make heuristic construction fast and cheap, we must take humans out of the loop. While techniques for automatic feature generation from the compiler IR have been proposed in the past [8, 9], they do not solve the problem in a practical way. They are deeply embedded into the compiler, require expert knowledge to guide the generation, have to be repeated from scratch for every new heuristic, and their search time can be prohibitive. Our insight was that such costly approaches are not necessary any more. Deep learning techniques have shown astounding successes in identifying complex patterns and relationships in images [10, 11], audio [12], and even computer code [7, 13, 14]. We hypothesized that deep neural networks should be able to automatically extract features from source code. Our experiments showed that even this was a conservative target: with deep neural networks we can bypass static feature extraction and learn optimization heuristics directly on raw code.

Figure 1b shows our proposed methodology. Instead of manually extracting features from input programs to generate training data, program code is used directly in the training data. Programs are fed through a series of neural networks which learn how code correlates with performance. Internally and without prior knowledge, the networks construct complex abstractions of the input program characteristics and correlations between those abstractions and performance. Our work replaces the need for compile-time or static code features, merging feature and heuristic construction into a single process of joint learning. Our system admits auxiliary features to describe information unavailable at compile time, such as the sizes of runtime input parameters. Beyond these optional inclusions, we are able to learn optimization heuristics without human supervision or guidance.

By employing *transfer learning* [15], our approach is able to produce high quality heuristics even when learning on a small number of programs. The properties of the raw code that are abstracted by the beginning layers of our neural networks are mostly independent of the optimization problem. We reuse these parts of the network across heuristics, and, in the process, we speed up learning considerably.

We evaluated our approach on two problems: heterogeneous device mapping and GPU thread coarsening. Good heuristics for these two problems are important for extracting performance from heterogeneous systems, and the fact that machine learning has been used before for heuristic construction for these problems allows direct comparison. Prior machine learning approaches resulted in good heuristics which extracted 73% and 79% of the available performance respectively but required extensive human effort to select the appropriate features. Nevertheless, our approach was able to outperform them by 14% and 12%, which indicates a better identification of important program characteristics, without any expert help.

We make the following contributions:

- We present a methodology for building compiler heuristics without any need for feature engineering.
- A novel tool DeepTune for automatically constructing optimization heuristics without features. DeepTune outperforms existing state-of-the-art predictive models by 14% and 12% in two challenging optimization domains.
- We apply, for the first time, *transfer learning* on compile-time and runtime optimizations, improving the heuristics by reusing training information across different optimization problems, even if they are completely unrelated.

2. DeepTune: Learning On Raw Program Code

DeepTune is an end-to-end machine learning pipeline for optimization heuristics. Its primary input is the source code of a program to be optimized, and through a series of neural networks, it directly predicts the optimization which should be applied. By learning on source code, our approach is not tied to a specific compiler, platform, or optimization problem. The same design can be reused to build multiple heuristics. The most important innovation of DeepTune is that it forgoes the need for human experts to select and tune appropriate features.

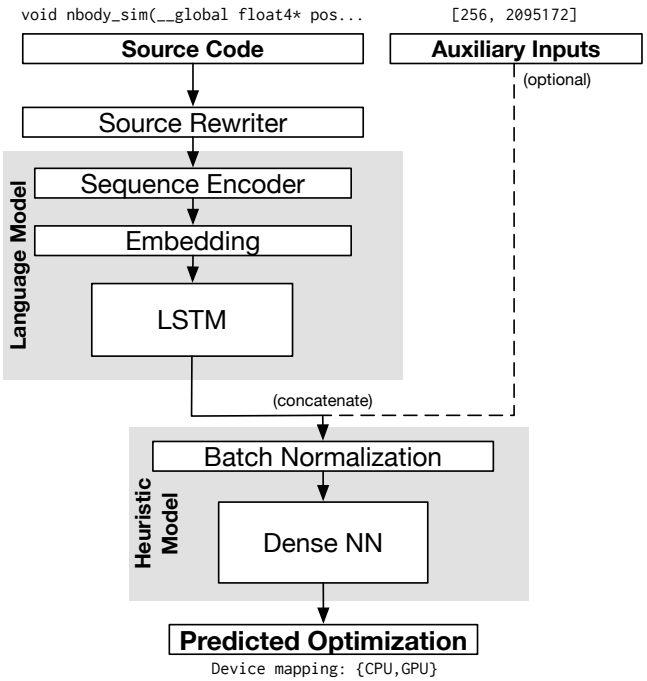


Figure 2: DeepTune architecture. Code properties are extracted from source code by the language model. They are fed, together with optional auxiliary inputs, to the heuristic model to produce the final prediction.

2.1. System Overview

Figure 2 provides an overview of the system. A source rewriter removes semantically irrelevant information (such as comments) from the source code of the target program and passes it to a language model. The language model converts the arbitrary length stream of code into a fixed length vector of real values which fully capture the properties and structure of the source, replacing the role of hand designed features. We then optionally concatenate this vector with auxiliary inputs, which allow passing additional data about runtime or architectural parameters to the model for heuristics which need more than just compile-time information. Finally, a standard feed-forward network is used to predict the best heuristic parameters to optimize the program.

DeepTune is open source[1]. We implemented the model using Keras, with TensorFlow [16] and Theano [17] backends.

2.2. Language Model

Learning effective representations of source code is a difficult task. A successful model must be able to:

- derive semantic and syntactic patterns of a programming language entirely from sample codes;

[1] DeepTune is available at: `http://chriscummins.cc/deeptune`

- identify the patterns and representation in source codes which are relevant to the task at hand; and
- discriminate performance characteristics arising from potentially subtle differences in similar codes.

To achieve this task, we employ state-of-the-art language modeling techniques, coupled with a series of generic, language agnostic code transformations.

Source Rewriter To begin with, we apply a series of *source normalizing* transformations, similar to those of Cummins *et al.* [7]. These transformations, implemented as an LLVM pass, parse the AST, removing conditional compilation, then rebuild the input source code using a consistent code style and identifier naming scheme. The role of source normalization is to simplify the task of modeling source code by ensuring that trivial semantic differences in programs such as the choice of variable names or the insertion of comments do not affect the learned model. Figures 3b and 3a show the source rewriting applied to a simple program.

Sequence Encoder We encode source code as a sequence of integers for interpretation by neural networks, where each integer is an index into a predetermined vocabulary. In [7], a character based vocabulary is used. This minimizes the size of the vocabulary, but leads to long sequences which are harder to extract structure from. In [18], a token based vocabulary is used. This leads to shorter sequences, but tokenizing real codes causes an explosion in the size of the vocabulary, as every identifier and literal must be represented uniquely.

We designed a hybrid, partially tokenized approach. This allows common multi-character sequences such as `float` and `if` to be represented as unique vocabulary items, but literals and other infrequently used words to be encoded at the character level.

We first assembled a candidate vocabulary V_c for the OpenCL programming language containing the 208 data types, keywords, and language builtins of the OpenCL specification. We then derived the subset of the candidate vocabulary $V \in V_c$ which is required to encode a corpus of 45k lines of handwritten GPGPU benchmark suite kernels. Beginning with the first character in the corpus, our 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 new program sources. Figure 3c shows the vocabulary derived for a single input source code Figure 3b.

Embedding During encoding, tokens in the vocabulary are mapped to unique integer values, e.g. `float` $\rightarrow 0$, `int` $\rightarrow 1$. The integer values chosen are arbitrary, and offer a *sparse* data representation, meaning that a language model cannot infer the relationships between tokens based on their integer values. This is in contrast to the *dense* representations of other domains, such as pixels used in computer vision, which can be interpolated between to derive the differences between colors.

```
1  //#define Elements
2  __kernel void memset_kernel(__global char * mem_d,
       ↪ short val, int number_bytes){
3      const int thread_id = get_global_id(0);
4      mem_d[thread_id] = val;
5  }
```

(a) An example, short OpenCL kernel, taken from Nvidia's *streamcluster*.

```
1  __kernel void A(__global char* a, short b, int c) {
2    const int d = get_global_id(0);
3    a[d] = b;
4  }
```

(b) The *streamcluster* kernel after source rewriting. Variable and function names are normalized, comments removed, and code style enforced.

idx	token	idx	token	idx	token
1	'__kernel'	10	','	19	'const'
2	' '	11	'short'	20	'd'
3	'void'	12	'b'	21	'='
4	'A'	13	'int'	22	'get_global_id'
5	'('	14	'c'	23	'0'
6	'__global'	15	')'	24	';'
7	'char'	16	'{'	25	'['
8	'*'	17	'\n'	26	']'
9	'a'	18	' '	27	'}'

(c) Derived vocabulary, ordered by their appearance in the input (b). The vocabulary maps tokens to integer indices.

01	02	03	02	04	05	06	02	07	08	02
09	10	02	11	02	12	10	02	13	02	14
15	02	16	17	18	19	02	13	02	20	02
21	02	22	05	23	15	24	17	18	09	25
20	26	02	21	02	12	24	17	27	<pad...>	

(d) Indices encoded kernel sequence. Sequences may be padded to a fixed length by repeating an out-of-vocabulary integer (e.g. -1).

Figure 3: Deriving a tokenized 1-of-k vocabulary encoding from an OpenCL source code.

To mitigate this, we use an *embedding*, which translates tokens in a sparse, integer encoded vocabulary into a lower dimensional vector space, allowing semantically related tokens like `float` and `int` to be mapped to nearby points [19, 20]. An embedding layer maps each token in the integer encoded vocabulary to a vector of real values. Given a vocabulary size V and embedding dimensionality D, an embedding matrix $\boldsymbol{W}_E \in \mathbb{R}^{V \times D}$ is learned during training, so that an integer encoded sequences of tokens $\boldsymbol{t} \in \mathbb{N}^L$ is mapped to the matrix $\boldsymbol{T} \in \mathbb{R}^{L \times D}$. We use an embedding dimensionality $D = 64$.

Sequence Characterization Once source codes have been encoded and translated into sequences of embedding vectors, neural networks are used to extract a fixed size vector which characterizes the entire source sequence. This is comparable to the hand engineered feature extractors used in existing approaches to predictive modeling, but is a *learned* process that occurs entirely— and automatically — within the hidden layers of the network.

We use the the Long Short-Term Memory (LSTM) architecture [21] for sequence characterization. LSTMs implements a Recurrent Neural Network in which the activations of neurons are learned with respect not just to their current inputs, but to previous inputs in a sequence. Unlike regular recurrent networks in which the strength of learning decreases over time (a symptom of the *vanishing gradients* problem [22]), LSTMs employ a *forget gate* with a linear activation function, allowing them to retain activations for arbitrary durations. This makes them effective at learning complex relationships over long sequences [23], an especially important capability for modeling program code, as dependencies in sequences frequently occur over long ranges (for example, a variable may be declared as an argument to a function and used throughout).

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.

2.3. Auxiliary Inputs

We support an arbitrary number of additional real valued *auxiliary inputs* which can be optionally used to augment the source code input. We provide these inputs as a means of increasing the flexibility of our system, for example, to support applications in which the optimization heuristic depends on dynamic values which cannot be statically determined from the program code [3, 24]. When present, the values of auxiliary inputs are concatenated with the output of the language model, and fed into a heuristic model.

2.4. Heuristic Model

The heuristic model takes the learned representations of the source code and auxiliary inputs (if present), and uses these values to make the final optimization prediction.

We first normalize the values. Normalization is necessary because the auxiliary inputs can have any values, whereas the language model activations are in the range [0,1]. If we did not normalize, then scaling the auxiliary inputs could affect the training of the heuristic model. Normalization occurs in batches. We use the normalization method of [25], in which each scalar of the heuristic model's inputs $x_1 \dots x_n$ is normalized to a mean 0 and standard deviation of 1:

$$x'_i = \gamma_i \frac{x_i - E(x_i)}{\sqrt{Var(x_i)}} + \beta_i$$

where γ and β are scale and shift parameters, learned during training.

The final component of DeepTune is comprised of two fully connected neural network layers. The first layer consists of 32 neurons. The second layer consists of a single neuron for each possible heuristic decision. Each neuron applies an activation function $f(x)$ over a weighted sum of its inputs. We use rectifier activation functions $f(x) = \max(0,x)$ for the first layer due to their improved performance during training of deep networks [26]. For the output layer, we use sigmoid activation functions $f(x) = \frac{1}{1+e^{-x}}$ which provide activations in the range $[0,1]$. The weights are learned during training.

The activation of each neuron in the output layer represents the model's confidence that the corresponding decision is the correct one. We take the arg max of the output layer to find the decision with the largest activation. For example, for a binary optimization heuristic the final layer will consist of two neurons, and the predicted optimization is the neuron with the largest activation.

2.5. Training the network

DeepTune is trained in the same manner as existing predictive model approaches, the key difference being that instead of having to manually create and extract features from programs, we simply use the raw program codes themselves.

The model is trained with Stochastic Gradient Descent (SGD), using the Adam optimizer [27]. For training data $X_1 \dots X_n$, SGD attempts to find the model parameters Θ that minimize the output of a loss function:

$$\Theta = \arg\min_{\Theta} \frac{1}{n} \sum_{i=1}^{n} \ell(X_i, \Theta)$$

where loss function $\ell(x, \Theta)$ computes the logarithmic difference between the predicted and expected values.

To reduce training time, multiple inputs are *batched* together and are fed into the neural network simultaneously, reducing the frequency of costly weight updates during backpropagation. This requires that the inputs to the language model be the same length. To achieve this, we pad all sequences up to a fixed length of 1024 tokens using a special padding token. This allows matrices of `batch_size` × `max_seq_len` tokens to be processed simultaneously. We note that batching and padding sequences to a maximum length is only to improve training time. Once deployed for prediction, sequences do not need to be padded, allowing classification of arbitrary length codes.

3. Experimental Methodology

We apply DeepTune to two heterogeneous compiler-based machine learning tasks and compare its performance to state-of-the-art approaches that use expert selected features.

3.1. Case Study A: OpenCL Heterogeneous Mapping

OpenCL provides a platform-agnostic framework for heterogeneous parallelism. This allows a program written in OpenCL to execute transparently across a range of different devices, from CPUs to GPUs and FPGAs. Given a program and a choice of execution devices, the question then is on which device should we execute the program to maximize performance?

State-of-the-art In [5], Grewe *et al.* develop a predictive model for mapping OpenCL kernels to the optimal device in CPU/GPU heterogeneous systems. They use supervised

Name		Description
F1:	`data size/(comp+mem)`	commun.-computation ratio
F2:	`coalesced/mem`	% coalesced memory accesses
F3:	`(localmem/mem)×wgsize`	ratio local to global mem accesses × #. work-items
F4:	`comp/mem`	computation-mem ratio

(a) Feature values

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

(b) Values used in feature computation

Table 1: Features used by Grewe *et al.* to predict heterogeneous device mappings for OpenCL kernels.

learning to construct decision trees, using a combination of static and dynamic kernel features. The static program features are extracted using a custom LLVM pass; the dynamic features are taken from the OpenCL runtime.

Expert Chosen Features Table 1a shows the features used by their work. Each feature is an expression built upon the code and runtime metrics given in Table 1b.

Experimental Setup We replicate the predictive model of Grewe *et al.* [5]. We replicated the experimental setup of [7] in which the experiments are extended to a larger set of 71 programs, summarized in Table 2a. The programs were evaluated on two CPU-GPU platforms, detailed in Table 3a.

DeepTune Configuration Figure 4a shows the neural network configuration of DeepTune for the task of predicting optimal device mapping. We use the OpenCL kernel source code as input, and the two dynamic values *workgroup size* and *data size* available to the OpenCL runtime.

Model Evaluation We use *stratified 10-fold cross-validation* to evaluate the quality of the predictive models [28]. Each program is randomly allocated into one of 10 equally-sized sets; the sets are balanced to maintain a distribution of instances from each class consistent with the full set. A model is trained on the programs from all but one of the sets, then tested on the programs of the unseen set. This process is repeated for each of the 10 sets, to construct a complete prediction over the whole dataset.

3.2. Case Study B: OpenCL Thread Coarsening Factor

Thread coarsening is an optimization for parallel programs in which the operations of two or more threads are fused together. This optimization can prove beneficial on certain combinations of programs and architectures, for example programs with a large potential for Instruction Level Parallelism on Very Long Instruction Word architectures.

	Version	#. benchmarks	#. kernels
NPB (SNU [29])	1.0.3	7	114
Rodinia [30]	3.1	14	31
NVIDIA SDK	4.2	6	12
AMD SDK	3.0	12	16
Parboil [31]	0.2	6	8
PolyBench [32]	1.0	14	27
SHOC [33]	1.1.5	12	48
Total	-	71	256

(a) Case Study A: OpenCL Heterogeneous Mapping

	Version	#. benchmarks	#. kernels
NVIDIA SDK	4.2	3	3
AMD SDK	3.0	10	10
Parboil [31]	0.2	4	4
Total	-	17	17

(b) Case Study B: OpenCL Thread Coarsening Factor

Table 2: Benchmark programs.

	Frequency	Memory	Driver
Intel Core i7-3820	3.6 GHz	8GB	AMD 1526.3
AMD Tahiti 7970	1000 MHz	3GB	AMD 1526.3
NVIDIA GTX 970	1050 MHz	4GB	NVIDIA 361.42

(a) Case Study A: OpenCL Heterogeneous Mapping

	Frequency	Memory	Driver
AMD HD 5900	725 MHz	2GB	AMD 1124.2
AMD Tahiti 7970	1000 MHz	3GB	AMD 1084.4
NVIDIA GTX 480	700 MHz	1536 MB	NVIDIA 304.54
NVIDIA K20c	706 MHz	5GB	NVIDIA 331.20

(b) Case Study B: OpenCL Thread Coarsening Factor

Table 3: Experimental platforms.

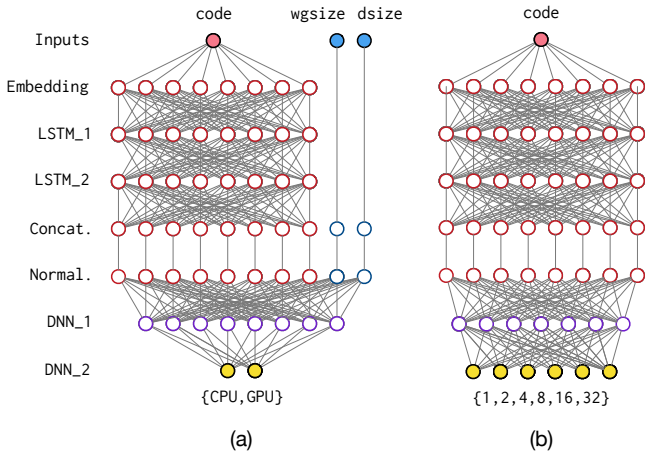
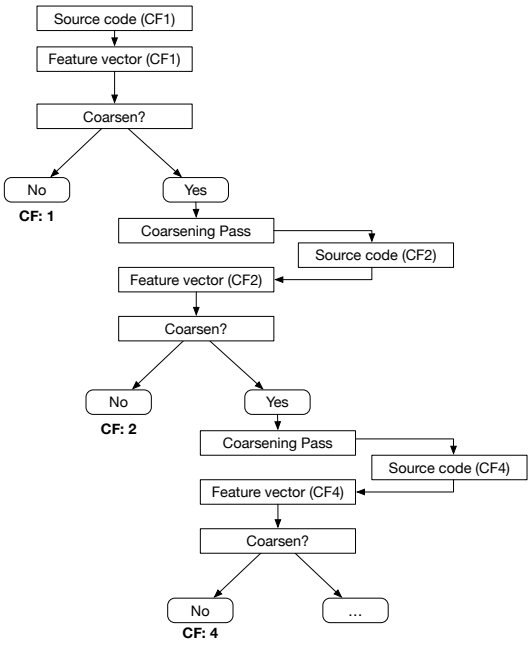
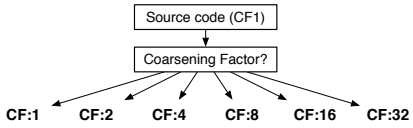


Figure 4: DeepTune neural networks, configured for (a) heterogeneous mapping, and (b) thread coarsening factor. The design stays almost the same regardless of the optimization problem. The only changes are the extra input for (a) and the number of nodes in the output layer.

State-of-the-art Magni *et al.* present a predictive model for OpenCL thread coarsening in [6]. They implement an iterative heuristic which determines whether a given program would



(a) Magni *et al.* cascading binary model.



(b) Our approach.

Figure 5: Two approaches for predicting coarsening factor (CF) of OpenCL kernels. Magni *et al.* reduce the multi-label classification problem to a series of binary decisions, by iteratively applying the optimization and computing new feature vectors. Our approach simply predicts the coarsening factor directly from the source code.

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, shown in Figure 5a. They select from one of six possible coarsening factors: $(1, 2, 4, 8, 16, 32)$, divided into 5 binary choices.

Expert Chosen Features Magni *et al.* followed a very comprehensive feature engineering process. 17 candidate features were assembled from a previous study of performance counters [34], and computed theoretical values [35]. For each candidate feature they compute its coarsening *delta*, reflecting the change in each feature value caused by coarsening: $f_\Delta = (f_{after} - f_{before})/f_{before}$, adding it to the feature set. Then they use Principle Component Analysis (PCA) on the 34 candidates and selected the first 7 principle components, accounting for 95% of the variance in the feature space.

Experimental Setup We replicate the experimental setup of Magni *et al.* [6]. The thread coarsening optimization is evaluated on 17 programs, listed in Table 2b. Four different GPU architectures are used, listed in Table 3b.

Name	Description
`BasicBlocks`	#. basic blocks
`Branches`	#. branches
`DivInsts`	#. divergent instructions
`DivRegionInsts`	#. instructions in divergent regions
`DivRegionInstsRatio`	#. instr. in divergent regions / total instructions
`DivRegions`	#. divergent regions
`TotInsts`	#. instructions
`FPInsts`	#. floating point instructions
`ILP`	average ILP / basic block
`Int/FP Inst Ratio`	#. branches
`IntInsts`	#. integer instructions
`MathFunctions`	#. match builtin functions
`MLP`	average MLP / basic block
`Loads`	#. loads
`Stores`	#. stores
`UniformLoads`	#. loads unaffected by coarsening direction
`Barriers`	#. barriers

Table 4: Candidate features used by Magni *et al.* for predicting thread coarsening. From these values, they compute relative deltas for each iteration of coarsening, then use PCA for selection.

	#. neurons		#. parameters	
	HM	CF	HM	CF
Embedding	64	64	8,256	8,256
LSTM_1	64	64	33,024	33,024
LSTM_2	64	64	33,024	33,024
Concatenate	64 + 2	-	-	-
Batch Norm .	66	64	264	256
DNN_1	32	32	2,144	2,080
DNN_2	2	6	66	198
Total			76,778	76,838

Table 5: The size and number of parameters of the DeepTune components of Figure 4, configured for heterogeneous mapping (HM) and coarsening factor (CF).

DeepTune Configuration Figure 4b shows the neural network configuration. We use the OpenCL kernel as input, and directly predict the coarsening factor.

Model Evaluation Compared to Case Study A, the size of the evaluation is small. We use *leave-one-out cross-validation* to evaluate the predictive models. For each program, a model is trained on data from all other programs and used to predict the coarsening factor of the excluded program.

Because [6] does not describe the parameters of the neural network, we perform an additional, *nested* cross-validation process to find the optimal network parameters for the Magni *et al.* model. For every program in the training set, we evaluate 48 combinations of network parameters. We select the best performing configuration from these 768 results to train a model for prediction on the excluded program. This nested cross-validation is repeated for each of the training sets. We do not perform this tuning of hyper-parameters for DeepTune.

3.3. Comparison of Case Studies

For the two different optimization heuristics, the authors arrived at very different predictive model designs, with very different features. By contrast, we take exactly the same approach for both problems. None of DeepTune's parameters were tuned for the case studies presented above. Their settings represent conservative choices expected to work reasonably well for most scenarios.

Table 5 shows the similarity of our models. The only difference between our network design is the auxiliary inputs for Case Study A and the different number of optimization decisions. The differences between DeepTune configurations is only two lines of code: the first, adding the two auxiliary inputs; the second, increasing the size of the output layer for Case Study B from two neurons to six. The description of these differences is larger than the differences themselves.

4. Experimental Results

We evaluate the effectiveness of DeepTune for two distinct OpenCL optimization tasks: predicting the optimal device to use to run a given OpenCL program, and predicting thread coarsening factors.

We first compare DeepTune against two expert-tuned predictive models, showing that DeepTune outperforms the state-of-the-art in both cases. We then show that by leveraging knowledge learned from training DeepTune for one heuristic, we can boost training for the other heuristic, further improving performance. Finally, we analyze the working mechanism of DeepTune.

4.1. Case Study 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. In this experiment, we compare our approach against a static single-device approach and the Grewe *et al.* predictive model. The *static mapping* selects the device which gave the best average case performance over all the programs. On the AMD platform, the best-performing device is the CPU; on the NVIDIA platform, it is the GPU.

Figure 6 shows the accuracy of both predictive models and the static mapping approach for each of the benchmark suites. The static approach is accurate for only 58.8% of cases on AMD and 56.9% on NVIDIA. This suggests the need for choosing the execution device on a per program basis. The Grewe *et al.* model achieves an average accuracy of 73%, a significant improvement over the static mapping approach. By automatically extracting useful feature representations from the source code, DeepTune gives an average accuracy of 82%, an improvement over the other two schemes.

Using the static mapping as a baseline, we compute the relative performance of each program using the device selected by the Grewe *et al.* and DeepTune models. Figure 7 shows these

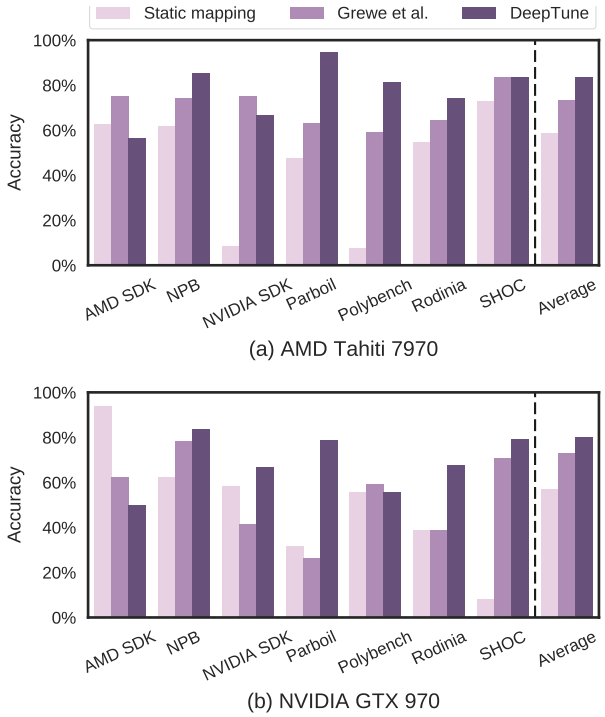


Figure 6: Accuracy of optimization heuristics for heterogeneous device mapping, aggregated by benchmark suite. The optimal static mapping achieves 58% accuracy. The Grewe *et al.* and DeepTune predictive models achieve accuracies of 73% and 84%, respectively.

speedups. Both predictive models significantly outperform the static mapping; the Grewe *et al.* model achieves an average speedup of 2.91× on AMD and 1.26× on NVIDIA (geomean 1.18×). In 90% of cases, DeepTune matches or outperforms the predictions of the Grewe *et al.* model, achieving an average speedup of 3.34× on AMD and 1.41× on NVIDIA (geomean 1.31×). This 14% improvement in performance comes at a greatly reduced cost, requiring no intervention by humans.

4.2. Case Study B: OpenCL Thread Coarsening Factor

Exploiting thread coarsening for OpenCL kernels is a difficult task. On average, coarsening slows programs down. The maximum speedup attainable by a perfect heuristic is 1.36×.

Figure 8 shows speedups achieved by the Magni *et al.* and DeepTune models for all programs and platforms. We use as baseline the performance of programs without coarsening. On the four experimental platforms (AMD HD 5900, Tahiti 7970, NVIDIA GTX 480, and Tesla K20c), the Magni *et al.* model achieves average speedups of 1.21×, 1.01×, 0.86×, and 0.94×, respectively. DeepTune outperforms this, achieving speedups of 1.10×, 1.05×, 1.10×, and 0.99×.

Some programs — especially those with large divergent regions or indirect memory accesses — respond very poorly to coarsening. No performance improvement is possible on the `mvCoal` and `spmv` programs. Both models fail to achieve

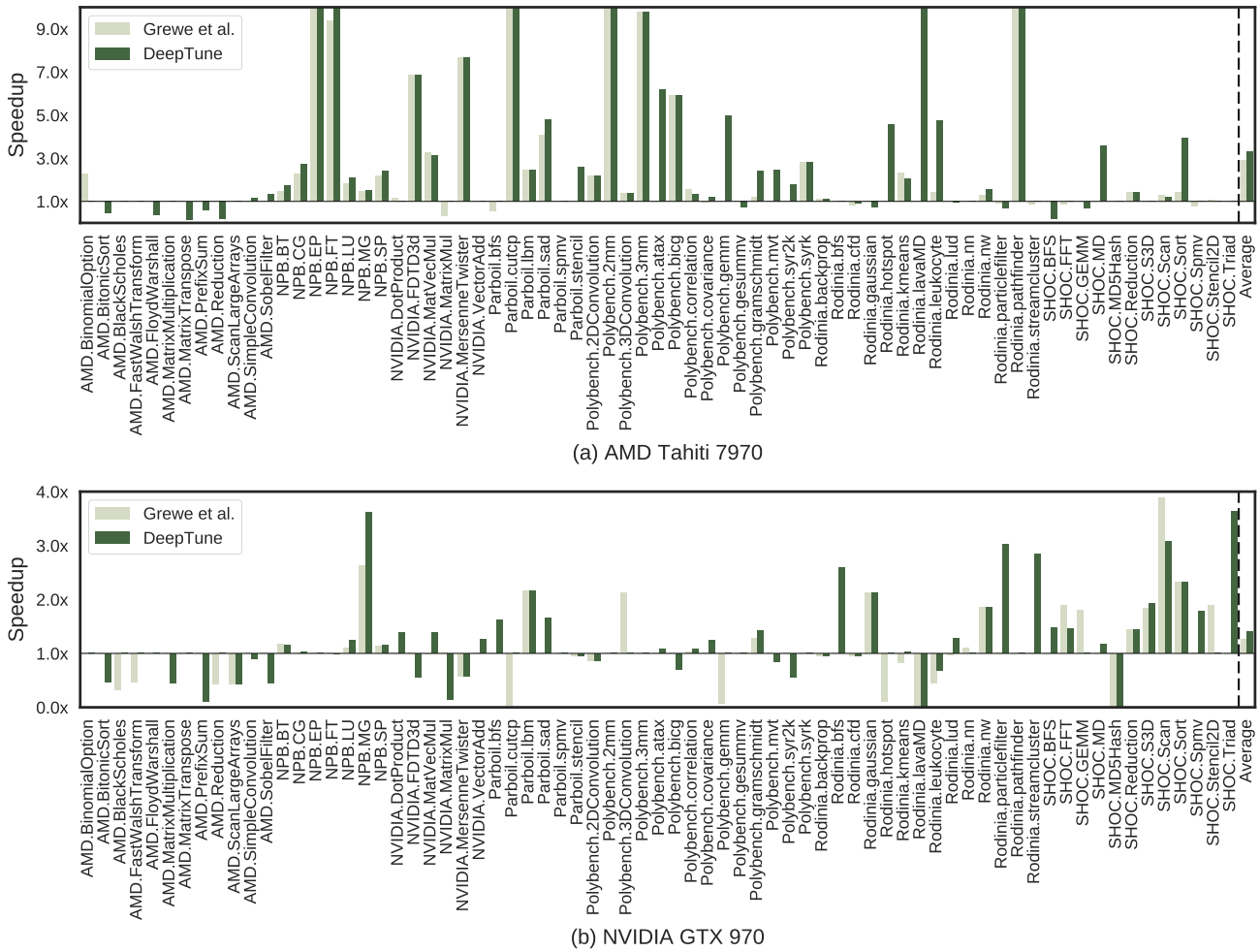


Figure 7: Speedup of predicted heterogeneous mappings over the best static mapping for both platforms. In (a) DeepTune achieves an average speedup of 3.43x over static mapping and 18% over Grewe *et al.* In (b) the speedup is 1.42x and 13% respectively.

positive average speedups on the NVIDIA Tesla K20c, because thread coarsening does not give performance gains for the majority of the programs on this platform.

The disappointing results for both predictive models can be attributed to the small training program set used by Magni *et al.* (only 17 programs in total). As a result, the models suffer from sparse training data. Prior research has shown that data sparsity can be overcome using additional programs; in the following subsection we describe and test a novel strategy for training optimization heuristics on a small number of programs by exploiting knowledge learned from other optimization domains.

4.3. Transfer Learning Across Problem Domains

There are inherent differences between the tasks of building heuristics for heterogeneous mapping and thread coarsening, evidenced by the contrasting choices of features and models in Grewe *et al.* and Magni *et al.* However, in both cases, the first role of DeepTune is to extract meaningful abstractions and representations of OpenCL code. Prior research in deep learning has shown that models trained on similar inputs for different tasks often share useful commonalities. The idea is that in neural network classification, information learned at the early layers of neural networks (i.e. closer to the input layer) will be useful for multiple tasks. The later the network layers are (i.e. closer to the output layer), the more specialized the layers become [36].

We hypothesized that this would be the case for DeepTune, enabling the novel transfer of information between models *across different optimization domains*. To test this, we extracted the language model — the `Embedding`, `LSTM_1`, and `LSTM_2` layers — trained for the heterogeneous mapping task and *transferred* it over to the new task of thread coarsening. Since DeepTune keeps the same design for both optimization problems, this is as simple as copying the learned weights of the three layers. Then we trained the model as normal.

As shown in Figure 8, our newly trained model, DeepTune-TL has improved performance for 3 of the 4 platforms: 1.17×,

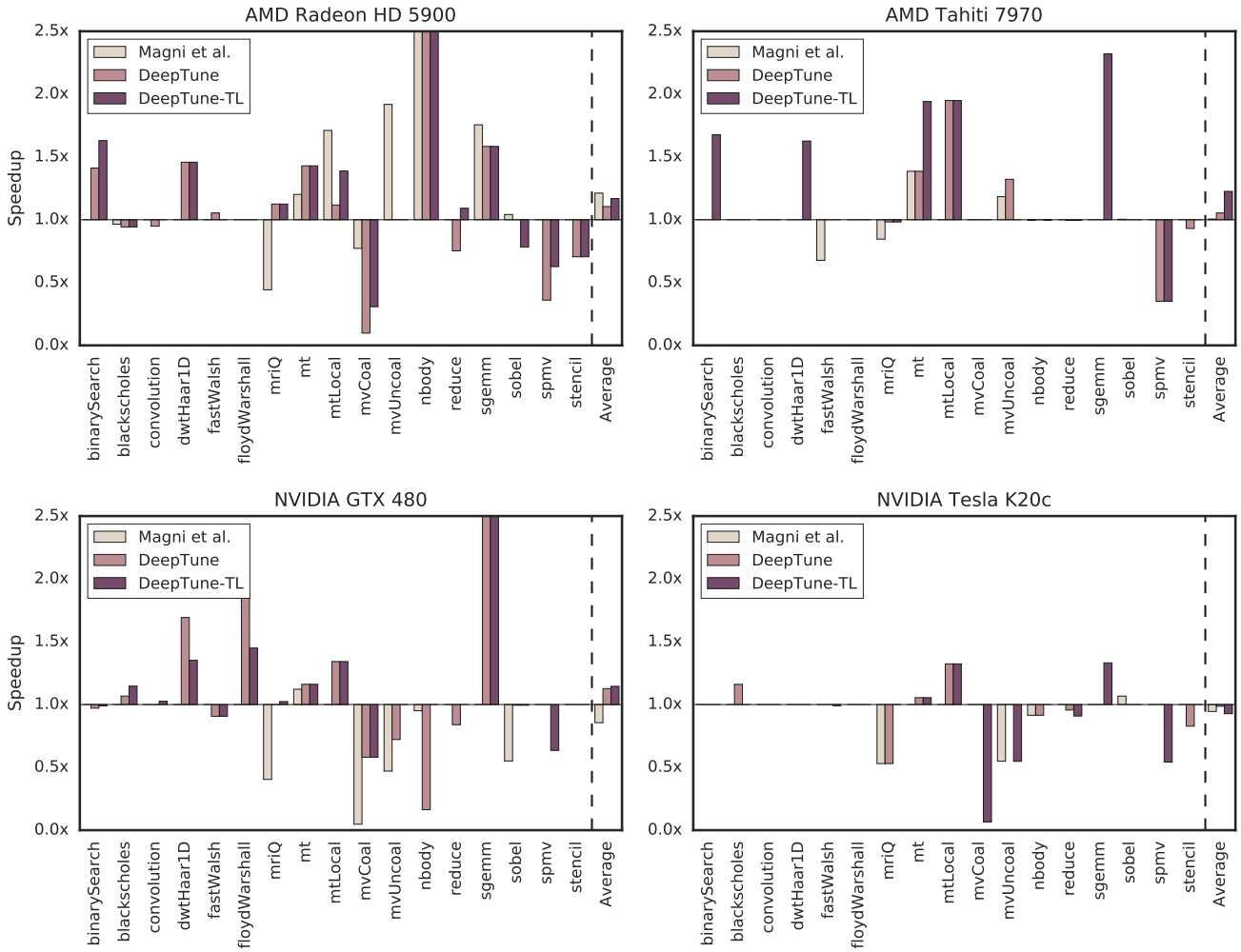


Figure 8: Speedups of predicted coarsening factors for each platform. DeepTune outperforms Magni *et al* on three of the four platforms. Transfer learning improves DeepTune speedups further, by 16% on average.

1.23×, 1.14×, 0.93× , providing an average 12% performance improvement over Magni *et al.* In 81% of cases, the use of transfer learning matched or improved the optimization decisions of DeepTune, providing up to a 16% improvement in per platform performance.

On the NVIDIA Tesla K20c, the platform for which no predictive model achieves positive average speedups, we match or improve performance in the majority of cases, but over-coarsening on three of the programs causes a modest reduction in average performance. We suspect that for this platform, further performance results are necessary due to its unusual optimization profile.

4.4. DeepTune Internal Activation States

We have shown that DeepTune automatically outperforms state-of-the-art predictive models for which experts have invested a great amount of time in engineering features. In this subsection we attempt to illuminate the inner workings, using a single example from Case Study B: predicting the thread coarsening factor for Parboil's `mriQ` benchmark on four different platforms.

Figure 9 shows the DeepTune configuration, with visual overlays showing the internal state. From top to bottom, we begin first with the input, which is the 267 lines of OpenCL code for the `mriQ` kernel. This source code is preprocessed, formatted, and rewritten using variable and function renaming, shown in Figure 9b. The rewritten source code is tokenized and encoded in a 1-of-*k* vocabulary. Figure 9c shows the first 80 elements of this encoded sequence as a heatmap in which each cell's color reflects its encoded value. The input, rewriting, and encoding is the same for each of the four platforms.

The encoded sequences are then passed into the Embedding layer. This maps each token of the vocabulary to a point in a 64 dimension vector space. Embeddings are learned during training so as to cluster semantically related tokens together. As such, they may differ between the four platforms. Figure 9d shows a PCA projection of the embedding space for one of the platforms, showing multiple clusters of tokens. By honing in

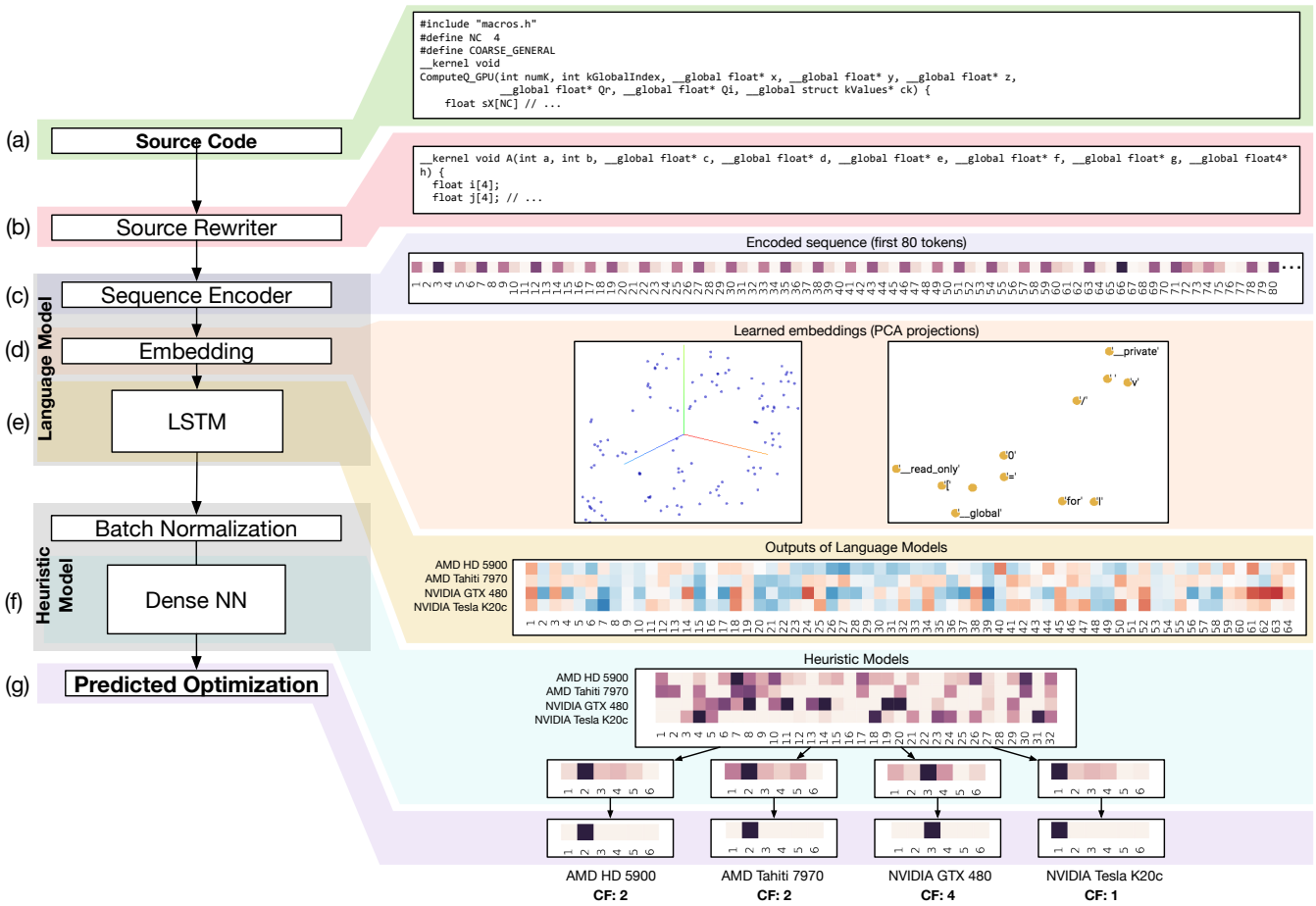


Figure 9: Visualizing the internal state of DeepTune when predicting coarsening factor for Parboil's `mriQ` benchmark on four different architectures. The activations in each layer of the four models increasingly diverge the lower down the network.

on one of the clusters and annotating each point with its corresponding token, we see that the cluster contains the semantically related OpenCL address space modifiers `__private`, `__global`, and `__read_only`.

Two layers of 64 LSTM neurons model the sequence of embeddings, with the neuron activations of the second layer being used to characterize the entire sequence. Figure 9e shows the neurons in this layer for each of the four platforms, using a red-blue heatmap to visualize the intensity of each activation. Comparing the activations between the four platforms, we note a number of neurons in the layer with different responses across platforms. This indicates that the language model is partly specialized to the target platform.

As information flows through the network, the layers become progressively more specialized to the specific platform. We see this in Figure 9f, which shows the two layers of the heuristic model. The activations within these increasingly diverge. The mean variance of activations across platforms increases threefold compared to the language model, from 0.039 to 0.107. Even the activations of the AMD HD 5900 and AMD Tahiti 7970 platforms are dissimilar, despite the final predicted coarsening factor for both platforms being the same. In Figure 9g we take the largest activation of the output layer as the final predicted coarsening factor. For this particular program, a state-of-the-art model achieves 54% of the maximum performance. DeepTune achieves 99%.

5. Related Work

Machine learning has emerged as a viable means in automatically constructing heuristics for code optimization [3, 4, 24, 37–39]. Its great advantage is that it can adapt to changing hardware platforms as it has no a priori assumptions about their behavior. The success of machine learning based code optimization has required having a set of high-quality features that can capture the important characteristics of the target program. Given that there is an infinite number of these potential features, finding the right set of features is a non-trivial, time-consuming task.

Various forms of program features have been used in compiler-based machine learning. These include static code structures [40] and runtime information such as system load [41] and performance counters [42]. In compiler research, the feature sets used for predictive models are often provided without explanation and rarely is the quality of those features evaluated. More commonly, an initial large, high dimensional

candidate feature space is pruned via feature selection [3], or projected into a lower dimensional space [43, 44]. FEAST employs a range of existing feature selection methods to select useful candidate features [45]. Unlike these approaches, DeepTune extracts features and reduces the dimensionality of the feature space completely internally and without expert guidance.

Park *et al.* present a unique graph-based approach for feature representations [46]. They use a Support Vector Machine where the kernel is based on a graph similarity metric. Their technique still requires hand coded features at the basic block level, but thereafter, graph similarity against each of the training programs takes the place of global features. Being a kernel method, it requires that training data graphs be shipped with the compiler, which may not scale as the size of the training data grows with the number of instances, and some training programs may be very large. Finally, their graph matching metric is expensive, requiring $O(n^3)$ to compare against each training example. By contrast, our method does not need any hand built static code features, and the deployment memory footprint is constant and prediction time is linear in the length of the program, regardless of the size of the training set.

A few methods have been proposed to automatically generate features from the compiler's intermediate representation [8, 9]. These approaches closely tie the implementation of the predictive model to the compiler IR, which means changes to the IR will require modifications to the model. The work of [9] uses genetic programming to search for features, and required a huge grammar to be written, some 160kB in length. Although much of this can be created from templates, selecting the right range of capabilities and search space bias is non trivial and up to the expert. The work of [8] expresses the space of features via logic programming over relations that represent information from the IRs. It greedily searches for expressions that represent good features. However, their approach relies on expert selected relations, combinators and constraints to work. For both approaches, the search time may be significant.

Cavazos *et al.* present a reaction-based predictive model for software-hardware co-design [47]. Their approach profiles the target program using several carefully selected compiler options to see how program runtime changes under these options for a given micro-architecture setting. They then use the program "reactions" to predict the best available application speedup. While their approach does not use static code features, developers must carefully select a few settings from a large number of candidate options for profiling, because poorly chosen options can significantly affect the quality of the model. Moreover, the program must be run several times before optimization, while our technique does not require the program to be profiled.

In recent years, machine learning techniques have been employed to model and learn from program source code on various tasks. These include mining coding conventions [14] and idioms [13], API example code [48] and pseudo-code generation [49], and benchmark generation [7]. Our work is the first attempt to extend the already challenging task of modeling distributions over source code to learning distributions over source code with respect to code optimizations.

Recently, deep neural networks [50] have been shown to be a powerful tool for feature engineering in various tasks including image recognition [10, 11] and audio processing [12]. In the field of compiler optimization, no work so far has applied deep neural networks for program feature generation and selection. Our work is the first to do so.

6. Conclusions

Applying machine learning to compile-time and runtime optimizations requires generating features first. This is a time consuming process, it needs supervision by an expert, and even then we cannot be sure that the selected features are optimal. In this paper we present a novel tool for building optimization heuristics, DeepTune, which forgoes feature extraction entirely, relying on powerful language modeling techniques to automatically build complex and effective representations of programs directly from raw source code. The result translates into a huge reduction in development effort, improved heuristic performance, and more simple model designs.

Our approach is fully automated. Using DeepTune, compiler developers no longer need to spend months using statistical methods and profile counters to select program features via trial and error. It is worth mentioning that we do not tailor our model design or parameters for the optimization task at hand, yet we achieve performance on par with and in most cases *exceeding* state-of-the-art predictive models.

We used DeepTune to automatically construct heuristics for two challenging optimization problems: selecting the optimal execution device for OpenCL kernels, and selecting OpenCL thread coarsening factors. In both cases, we outperform state-of-the-art predictive models, achieving performance improvements of 16% and 12%, respectively. We have also shown that the DeepTune architecture allows us to exploit information learned from another optimization problem to give the learning a boost. Doing so provides up to a 16% performance improvement when training using a handful of training programs. We suspect that this approach will be useful for other optimization tasks for which training programs are a scarce resource.

In future work, we will extend our heuristic construction approach by automatically learning dynamic features over raw data; apply unsupervised learning techniques [51] over unlabeled source code to further improve learned representations of programs; and deploy trained DeepTune heuristic models to low power embedded systems using optimization and compression of neural networks [52].

References

[1] P. Micolet, A. Smith, and C. Dubach. "A Machine Learning Approach to Mapping Streaming Workloads to Dynamic Multicore Processors." In: *LCTES*. ACM, 2016.

[2] T. L. Falch and A. C. Elster. "Machine Learning Based Autotuning for Enhanced OpenCL Performance Portability." In: *IPDPSW*. IEEE, 2015.

[3] M. Stephenson and S. Amarasinghe. "Predicting unroll factors using supervised classification." In: *CGO*. IEEE, 2005.

[4] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. "Using Machine Learning to Focus Iterative Optimization." In: *CGO*. IEEE, 2006.

[5] D. Grewe, Z. Wang, and M. O'Boyle. "Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems." In: *CGO*. IEEE, 2013.

[6] A. Magni, C. Dubach, and M. O'Boyle. "Automatic Optimization of Thread-Coarsening for Graphics Processors." In: *PACT*. ACM, 2014.

[7] C. Cummins, P. Petoumenos, W. Zang, and H. Leather. "Synthesizing Benchmarks for Predictive Modeling." In: *CGO*. IEEE, 2017.

[8] M. Namolaru, A. Cohen, G. Fursin, A. Zaks, and A. Freund. "Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization." In: *CASES*. 2010.

[9] H. Leather, E. Bonilla, and M. O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation." In: *CGO*. IEEE, 2009.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: *NIPS*. 2012.

[11] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." In: *CVPR*. IEEE, 2016.

[12] H. Lee, Y. Largman, P. Pham, and A. Y. Ng. "Unsupervised Feature Learning for Audio Classification using Convolutional Deep Belief Networks." In: *NIPS*. 2009.

[13] M. Allamanis and C. Sutton. "Mining Idioms from Source Code." In: *FSE*. ACM, 2014.

[14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. "Learning Natural Coding Conventions." In: *FSE*. ACM, 2014.

[15] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. "How Transferable are Features in Deep Neural Networks?" In: *NIPS*. 2014.

[16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A system for large-scale machine learning." In: *arXiv:1605.08695* (2016).

[17] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, and Y. Bengio. "Theano: Deep Learning on GPUs with Python." In: *BigLearning Workshop*. 2011.

[18] M. Allamanis and C. Sutton. "Mining Source Code Repositories at Massive Scale using Language Modeling." In: *MSR*. 2013.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean. "Distributed Representations of Words and Phrases and their Compositionality." In: *NIPS*. 2013.

[20] M. Baroni, G. Dinu, and G. Kruszewski. "Don't Count, Predict! A Systematic Comparison of Context-Counting vs . Context-Predicting Semantic Vectors." In: *ACL*. 2014.

[21] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (1997).

[22] R Pacanu, T Mikolov, and Y Bengio. "On the Difficulties of Training Recurrent Neural Networks." In: *ICML*. 2013.

[23] Z. C. Lipton, J. Berkowitz, and C. Elkan. "A Critical Review of Recurrent Neural Networks for Sequence Learning." In: *arXiv:1506.00019* (2015).

[24] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U. O'Reilly, and S. Amarasinghe. "Autotuning Algorithmic Choice for Input Sensitivity." In: *PLDI*. ACM, 2015.

[25] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *arXiv:1502.03167* (2015).

[26] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines." In: *ICML*. 2010.

[27] D. P. Kingma and J. L. Ba. "Adam: a Method for Stochastic Optimization." In: *ICLR* (2015).

[28] J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Elsevier, 2011.

[29] S. Seo, G. Jo, and J. Lee. "Performance Characterization of the NAS Parallel Benchmarks in OpenCL." In: *IISWC*. IEEE, 2011.

[30] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. "Rodinia: A Benchmark Suite for Heterogeneous Computing." In: *IISWC*. IEEE, Oct. 2009.

[31] J. A. Stratton, C. Rodrigues, I. Sung, N. Obeid, L. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing." In: *Center for Reliable and High-Performance Computing* (2012).

[32] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. "Auto-tuning a High-Level Language Targeted to GPU Codes." In: *InPar*. 2012.

[33] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite." In: *GPGPU*. ACM, 2010.

[34] A. Magni, C. Dubach, and M. O'Boyle. "A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening." In: *SC*. 2013.

[35] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications." In: *PPoPP*. ACM, 2012.

[36] M. D. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks." In: *ECCV*. 2014.

[37] Z. Wang and M. O'Boyle. "Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach." In: *PACT*. ACM, 2010.

[38] Sameer Kulkarni and John Cavazos. "Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning." In: *OOPSLA*. ACM, 2012.

[39] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai. "Architecture-Adaptive Code Variant Tuning." In: *ASPLOS*. ACM, 2016.

[40] Y. Jiang, Z. Z. Zhang, K. Tian, F. Mao, M. Gethers, X. Shen, and Y. Gao. “Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors.” In: *CGO* (2010).

[41] Y. Wen, Z. Wang, and M. O’Boyle. “Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms.” In: *HiPC*. IEEE, 2014.

[42] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. O’Boyle. “Portable Compiler Optimisation Across Embedded Programs and Microarchitectures using Machine Learning.” In: *MICRO*. ACM, 2009.

[43] A. Collins, C. Fensch, H. Leather, and M. Cole. “MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons.” In: *HiPC*. IEEE, 2013.

[44] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. O’Boyle, and O. Temam. “Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction.” In: *CF*. ACM, 2007.

[45] P. Ting, C. Tu, Pi. Chen, Y. Lo, and S. Cheng. “FEAST: An Automated Feature Selection Framework for Compilation Tasks.” In: *arXiv:1610.09543* (2016).

[46] E. Park, J. Cavazos, and M. A. Alvarez. “Using Graph-Based Program Characterization for Predictive Modeling.” In: *CGO*. IEEE, 2012.

[47] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O’Boyle, G. Fursin, and O. Temam. *Automatic Performance Model Construction for the Fast Software Exploration of New Hardware Designs*. 2006.

[48] X. Gu, H. Zhang, D. Zhang, and S. Kim. “Deep API Learning.” In: *FSE*. ACM, 2016.

[49] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura. “Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T).” In: *ASE*. IEEE, 2015.

[50] Y. LeCun, Y. Bengio, and G. Hinton. “Deep learning.” In: *Nature* 521.7553 (2015).

[51] Q. V. Le, R. Monga, M. Devin, G. Corrado, K. Chen, M. A. Ranzato, J. Dean, and A. Y. Ng. “Building High-level Features Using Large Scale Unsupervised Learning.” In: *ICML*. 2012.

[52] S. Han, H. Mao, and W. J. Dally. “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding.” In: *arXiv:1510.00149* (2015).

A. Artifact description

A.1. Abstract

Our research artifact consists of interactive Jupyter notebooks. The notebooks enable users to replicate all experiments in the paper, evaluate results, and plot figures.

A.2. Description

A.2.1. Check-list (Artifact Meta Information)

- **Run-time environment:** Ubuntu Linux and a web browser.
- **Hardware:** Users with an NVIDIA GPU may enable CUDA support to speed up computation of experiments.
- **Output:** Trained neural networks, predictive model evaluations, figures and tables from the paper.
- **Experiment workflow:** Install and run Jupyter notebook server; interact with and observe results in web browser.
- **Experiment customization:** Edit code and parameters in Jupyter notebooks.
- **Publicly available?:** Yes, code and data. See:
  `https://chriscummins.cc/pact17/`

A.2.2. How Delivered A publicly available git repository containing Jupyter notebooks and experimental data.

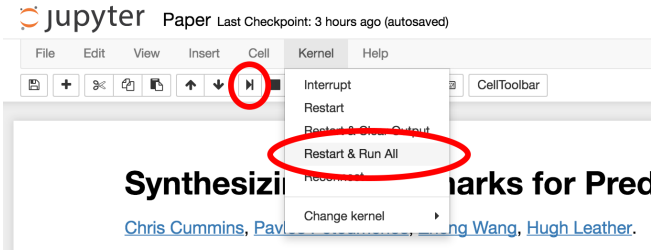
A.3. Installation

See `https://chriscummins.cc/pact17/` for instructions. The `code` directory contains the Jupyter notebooks. Following the build instructions described in `code/README.md`, the full installation process is:

```
$ ./bootstrap.sh | bash
$ ./configure
$ make
```

A.4. Experiment Workflow

1. Launch the Jupyter server using the command: `make run`.
2. In a web browser, navigate to `http://localhost:8000`.
3. Select a Jupyter notebook to open it.
4. Repeatedly press the *play* button (tooltip is "run cell, select below") to step through each cell of the notebook.
   OR select "Kernel" > "Restart & Run All" from the menu to run all of the cells in order.



A.5. Evaluation and Expected Result

Code cells within Jupyter notebooks display their output inline, and may be compared against the values in the paper. Expected results are described in text cells.

A.6. Experiment Customization

The experiments are fully customizable. The Jupyter notebook can be edited "on the fly". Simply type your changes into the cells and re-run them. For example,
Note that some of the code cells depend on the values of prior cells, so must be executed in sequence. Select "Kernel" > "Restart & Run All" from the menu to run all of the cells in order.

A.7. Notes

For more information about DeepTune, visit:
`https://chriscummins.cc/deeptune`
For more information about Artifact Evaluation, visit:
`http://ctuning.org/ae`