BenchDirect: A Directed Language Model for Compiler Benchmarks

Foivos Tsimpourlas
Meta AI Research
University of Edinburgh
F.Tsimpourlas@sms.ed.ac.uk

Pavlos Petoumenos
University of Manchester
pavlos.petoumenos@manchester.ac.uk

Min Xu
Meta AI Research
m1n@fb.com

Chris Cummins
Meta AI Research
cummins@fb.com

Kim Hazelwood
Meta AI Research
kimhazelwood@fb.com

Ajitha Rajan
University of Edinburgh
arajan@inf.ed.ac.uk

Hugh Leather
Meta AI Research
hleather@fb.com

ABSTRACT

The exponential increase of hardware-software complexity has made it impossible for compiler engineers to find the right optimization heuristics manually. Predictive models have been shown to find near optimal heuristics with little human effort but they are limited by a severe lack of diverse benchmarks to train on. Generative AI has been used by researchers to synthesize benchmarks into existing datasets. However, the synthetic programs are short, exceedingly simple and lacking diversity in their features.

We develop BenchPress, the first ML compiler benchmark generator that can be directed within source code feature representations. BenchPress synthesizes executable functions by infilling code that conditions on the program's left and right context. BenchPress uses active learning to introduce new benchmarks with unseen features into the dataset of Grewe's et al. CPU vs GPU heuristic, improving its acquired performance by 50%. BenchPress targets features that has been impossible for other synthesizers to reach. In 3 feature spaces, we outperform human-written code from GitHub, CLgen, CLSmith and the SRCIROR mutator in targeting the features of Rodinia benchmarks.

BenchPress steers generation with beam search over a feature-agnostic language model. We improve this with BenchDirect which utilizes a directed LM that infills programs by jointly observing source code context and the compiler features that are targeted. BenchDirect achieves up to 36% better accuracy in targeting the features of Rodinia benchmarks, it is 1.8× more likely to give an exact match and it speeds up execution time by up to 72% compared to BenchPress. Both our models produce code that is difficult to distinguish from human-written code. We conduct a Turing test which shows our models' synthetic benchmarks are labelled as 'human-written' as often as human-written code from GitHub.

Our source code is publicly available at https://github.com/fivosts/BenchPress. This work was supported by the Engineering and Physical Sciences Research Council (grant EP/L01503X/1), EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics. This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

PACT '22, October 10–12, 2012, Chicago,IL
2018. ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00

ACM Reference Format:
Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. 2018. BenchDirect: A Directed Language Model for Compiler Benchmarks. In *PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT), October 10–12, 2018, Chicago, IL.* ACM, New York, NY, USA, 18 pages.

1 INTRODUCTION

Predictive modeling for compiler optimisation heuristics has been shown to outperform human experts and reduce development time in previous studies [5, 7, 34, 36]. Predictive models learn such heuristics by training on source-level benchmarks or on static code features extracted at the (1) syntax level by traversing their Astract Syntax Tree (AST) or (2) Intermediate Representation (IR) with the help of compiler passes, as shown in Figure 1. However, predictive modeling's effectiveness is restricted by an acute shortage of benchmarks, both in quantity and feature diversity [7, 35, 38], degrading their performance.

There have been some recent generative approaches that leverage the rise of deep learning and language modeling to mitigate this shortage by automatically generating synthetic programs to enhance existing human-written benchmarks [1, 5, 7]. While they could provide elegant solutions to improve training data for predictive models, these synthetic benchmarks seem to be short, repetitive with little new features compared to existing benchmarks [14]. To generate programs, they either use static programming language specifications with fuzzing or sample programs from learnt distributions, e.g., machine learning algorithms. Their common characteristic is that they generate random benchmarks that are likely to conform to the language's grammar but they are highly unlikely to synthesize benchmarks that are both human-likely and are not already included in existing datasets. What is needed is a systematic method to search for missing programs whose features would be likely to improve the performance of trained downstream tasks. We aim to address this with BenchPress, a targeted benchmark generator, that can generate compiler benchmarks with a desired set of features. In this work, we focus on generating OpenCL benchmarks, as predictive models for heterogeneous systems is a rapidly advancing field and training examples for them are very sparse.

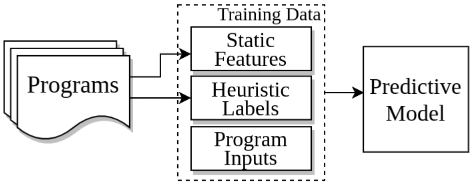


Figure 1: Training pipeline of a predictive model.

We develop BenchPress [12], a BERT-based OpenCL benchmark generator [9, 32] that targets and synthesizes benchmarks in desired parts of the feature space. We use active learning to choose parts of the feature space and beam search to steer BenchPress's generated samples towards the requested features. We train BenchPress with OpenCL code samples that we collect by mining BigQuery [15] and GitHub directly using its API [13]. We support composite data types and calls to user-defined functions in our dataset and benchmark generation. BenchPress is a bidirectional generative model and learns to generate code in any part of a sequence by jointly considering left and right context. We achieve this with a new learnt token, the [HOLE], which hides a sequence from the input, whose length is unknown to BenchPress during training. BenchPress learns to fill [HOLE] by iteratively predicting an arbitrary number tokens that are likely to lead to a compiling function. We further develop BenchDirect, an extension of BenchPress with a synthesizer conditioned on the features of the complete function. At inference time, this allows us to fill each [HOLE] with code that is more likely to bring us closer to the requested features.

BenchPress outperforms CLgen in the task of undirected program generation from a fixed input feed, generating 10× more unique OpenCL kernels that are 7.5× longer on average, with a compilation rate of 86% compared to CLgen's 2.33%. BenchPress strongly outperforms benchmark synthesizers CLgen, CLSmith [1, 39], and human written code from GitHub in reaching close to the features of Rodinia benchmarks, developed by compiler experts. The extended synthesizer, by directly filling holes with code that is useful for reaching the targeted features, makes this process 6% up to 72% faster, 6% up to 36% more accurate and 1.8× more likely to perfectly reach these features. Finally, BenchPress uses active learning, specifically query by committee [30], to search the feature space and find missing features to improve Grewe's et al. [16] CPU vs GPU heuristic. Enhancing the heuristic's dataset with BenchPress's benchmarks improves the heuristic's speedup relative to the optimal static decision by 50%, increasing it from 4% to 6%, when the maximum possible speedup for this task is 12%.

In this paper, we present the following contributions:

(1) We are the first to develop a feature-space agnostic, directed code generator towards desired program features.
(2) We develop an automated approach to rank the feature space of downstream tasks with active learning.
(3) We enable bidirectional source code generation by inserting [HOLE] tokens in any part of a sequence.

1.1 New Contributions

The contributions of this study, different from our previous work, are summarized as follows:

(1) We develop BenchDirect, the first bi-directional language model for code infilling that is directed in compiler feature spaces. Compared to BenchPress's language model's random benchmark generation, BenchDirect jointly conditions on code context and target features to generate directly candidates that satisfy them. We conduct an extensive evaluation between BenchPress and BenchDirect and we show the latter develops up to 36% better accuracy in targeting the features of Rodinia benchmarks across 3 feature spaces, while at the same time it requires up to 72% less time.
(2) We evaluate the human-likeness of BenchPress's, BenchDirect's, CLgen's and CLSmith's benchmarks as a means to measure their quality. We find benchmarks generated by BenchPress and BenchDirect to be 'human-written' labelled as often as code from GitHub from participants in a Turing test.

2 MOTIVATION

Figure 2 shows a two-dimensional slice of the Grewe's et al. [16] feature space: number of computational instructions vs number of memory instructions. Figure 2 also shows how the OpenCL benchmarks found in the Rodinia suite map into this plane, represented as purple diamonds. We find much of this two dimensional space is uncovered. 54 of the 58 Rodinia examples cluster in the lower left corner, the rest of the space having only four examples. Any optimization decision for programs in this area of the space would not be accurate due to lack of representative examples.

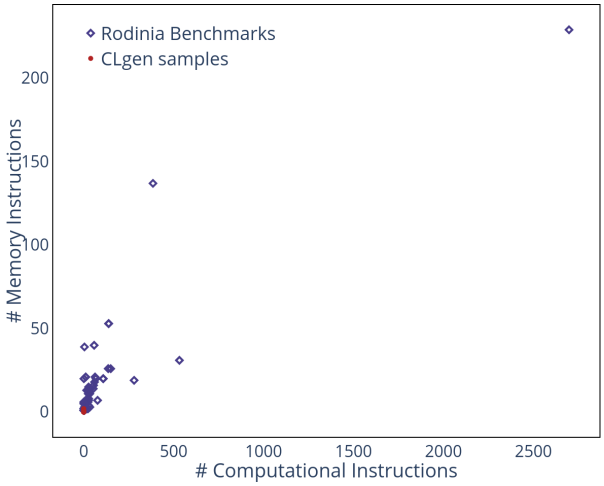


Figure 2: # Memory operations and # computational instructions for (a) Rodinia benchmarks in purple diamonds and (b) CLgen's samples in red dots. Generating samples with missing features is vital for predictive modeling's performance.

CLgen attempted to address this problem by automatically generating more training examples. However, the generated kernels lacked feature diversity and provided even poorer coverage of the feature space. Figure 2 represents their position in the 2D space as red dots. Almost all of them are concentrated in a corner covering a small percentage of the feature space. While CLgen can generate hundreds of millions of unique kernels, almost all of them will fail to compile. As the probability of having at least one illegal token in the kernel body increases with the number of tokens, only tiny kernels are valid. In our experiments in Section 5, the longest compiling CLgen kernel had 8 lines and 102 tokens. Given the small number of tokens in valid kernels, there is a high degree of repetitiveness in the generated corpus, not only in terms of features but also in terms of structure and functionality. As a result, this approach is not well suited to augmenting the training set with diverse feature benchmarks. There is a compelling need to generate training points for uncovered regions of the feature space and we attempt to address this need with BenchPress. In the following Sections, we discuss our approach and evaluation of BenchPress, comparing it to the existing state-of-the art for feature space coverage.

3 APPROACH

We present BenchPress, a deep learning model for directed compiler benchmark generation. BenchPress is the first directed synthesizer for compiling functions with features targeted by a user or a downstream task. BenchPress consists of an undirected language model that is trained on source code and a beam search sampler that steers its generation. Given a downstream task, our model uses active learning to search desired features and direct its program generation towards areas of high importance for the task. We further extend BenchPress's underlying language model into a directed synthesizer by encoding compiler features into the model's training process. This enables token generation to attend directly on the targeted features, significantly optimising steerable synthesis. We name this architecture BenchDirect.

BenchPress and BenchDirect share a BERT-based language model [9], which we transform into a generative model. There are two key features in our language model that enable directed, bi-directional program generation. First, we develop a new token, namely, the [HOLE], and we train BenchPress to iteratively fill holes of unknown length at any part of an input sequence by conditioning it on the left and right context of the [HOLE]. As an extension to this, BenchDirect's language model includes a Transformer-based encoder [37] that incorporates target compiler features into token classification. This allows tokens to be selected not only with respect to the input's source code context, but also given the compiler features that are targeted.

Figure 3 illustrates an overview of our approach. BenchPress consists of three main components:

(1) Learning corpus collection and processing.
(2) Directed source code language modeling.
(3) Feature space search and benchmark generation.

We discuss each step in the following four subsections. In our last subsection, we discuss BenchDirect's directed language model, which distinguishes it from our base architecture, BenchPress. Our codebase and experimental data are publicly available [1] for researchers to use.

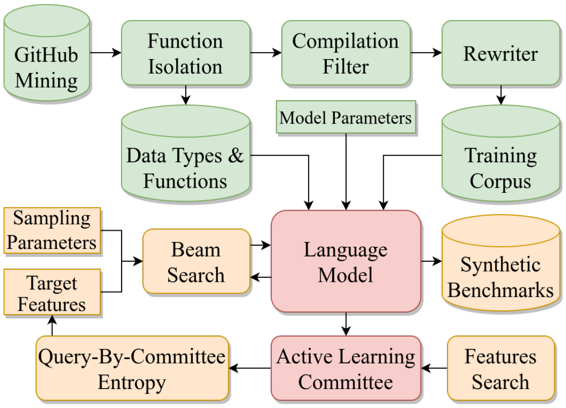


Figure 3: BenchPress's high-level approach.

3.1 Learning Corpus

Modeling source code accurately requires large amounts of data [24] similarly to other deep learning tasks. We develop a tool to collect data from BigQuery's GitHub dataset [15]. We also use GitHub's API [13] and mine directly extra repositories that are not included in BigQuery.

There are a few innovations in how we pre-process the code compared to previous works. First, we inline included header files recursively into source files to resolve type dependencies. Additionally, we automatically extract custom data types (e.g. struct, typedef) and utility functions found in the unprocessed corpus and place them into header files that are accessible throughout BenchPress's pipeline. This way, we resolve most type dependencies by retaining the functionality and semantics of the original, human-written programs. These two steps enable us to increase significantly the amount of compiling kernels we end up with in our training dataset. Second, we isolate kernels into single instances because BenchPress is trained on complete functions. From the previous steps, the type dependencies of each kernel are known and we automatically provide them to the compiler, retaining their compilability. Finally, we compile all kernels with Clang and reject those that do not compile.

Next, we re-write identifiers by randomly sampling the alphabet, eliminating spurious naming patterns in the corpus. All kernels are padded to BenchPress's sequence length and kernels that are longer than this are truncated to fit. This helps BenchPress train its later indices' positional embeddings more effectively, for which we have less training information compared to earlier indices. Finally, we derive a tokenizer by parsing the AST of all source code. We reserve tokens for all OpenCL keywords and all intrinsic OpenCL function name identifiers found in the official OpenCL specifications [31]. We analyze the dataset and tokenize by word the most common function names and custom data type identifiers that we have collected. We encode all literals and infrequently used custom types and functions character by character to avoid exploding the size of the vocabulary. We define 5 meta tokens: [START], [END], [PAD], [HOLE], [ENDHOLE]. The derived tokenizer holds in total 2,201 unique tokens.

[1] https://github.com/fivosts/BenchPress

3.2 Language Modeling

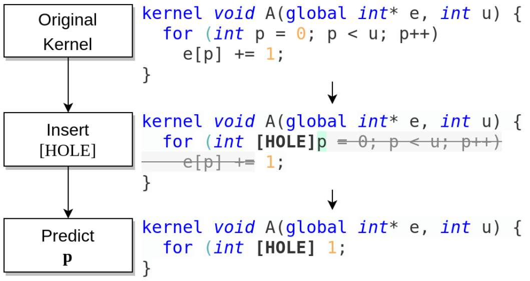
BenchPress is based on BERT [9], a Transformer-based model originally designed for natural language modeling. BERT is trained to predict words that have been randomly hidden by [MASK] tokens. This way BERT learns fitting words with respect to their position in a sequence and also the left and right context, i.e., the text sequence before and after the masked token to be predicted. This type of training helps BERT learn what words mean within a given context, improving downstream tasks that rely on that knowledge.

While this is a useful property, it is not enough to turn BERT into a generative model. We also want to be able to extend a kernel by inserting an arbitrary number of tokens in arbitrary positions. We could iteratively add a [MASK] token to get one extra token at a time, until we have a full statement. This would be limiting. Each time the new token would be selected based on its probability of completing forming a plausible kernel. Every intermediate kernel in the iterative process would have to be plausible or almost plausible, which is not a general way for augmenting kernels.

Clusters of [MASK] tokens could allow us to insert multiple tokens in each iteration. This is still unsatisfactory. The number of [MASK] tokens in the cluster biases the kind of code that will be generated: if we ask such a generator to produce five tokens, it will give us a five token statement that could be expected to close this gap, not a five token sequence that could be the start of a much longer statement. We could place the left and right context to the edges of a sequence and fill intermediate positions with [MASK] tokens. BenchPress could predict a vocabulary or a stop token for a [MASK], allowing for arbitrary sequences. We test this configuration and sample a trained model with a fixed input feed. BenchPress is unable to learn the [MASK]s' left and right context conditionally, when many [MASK]s are in a sequence, which leads to zero samples to compile or even resemble reasonable code.

What we do instead is to extend BERT's functionality with a new pair of learnt tokens, the [HOLE] and the [ENDHOLE]. [HOLE] follows the same logic with [MASK], however the number of tokens that have been hidden behind it is unknown to the model during training. The model only learns to predict the first token of an arbitrarily long missing sequence. At inference-time, we iteratively predict the first token of the remaining sequence and re-insert it just before the [HOLE]. This way BenchPress learns to generate arbitrarily large code sequences within any part of a sequence.

Figure 4 shows how a [HOLE] is inserted into a function to create a datapoint. A random starting index and a random length are selected. The choice of index and length are only restricted by a potential overlap of the prospective hidden sequence with any of the other meta token or the maximum hole length that is defined as a training parameter for the architecture as a percentage of each function's length. When the specifications of a hole have been settled, the hidden sequence is discarded. Only the first token of it is kept as the target prediction for that hole. A hole can also represent an empty sequence, i.e. hiding 0 tokens. In this case, the target prediction during training is [ENDHOLE]. The training instances are randomly generated on demand, the entire space of possible instances is too large to be pre-generated. In this paper, we only insert 1 hole per training instance for BenchPress to learn. Multiple holes could be used during training, but this is not needed during BenchPress's current benchmark generation task.



```
kernel void A(global int* e, int u) {
  for (int p = 0; p < u; p++)
    e[p] += 1;
}
```

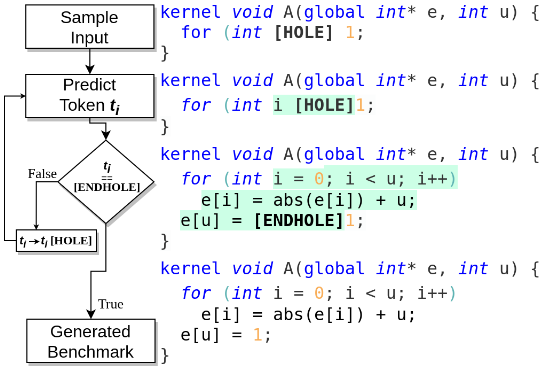
```
kernel void A(global int* e, int u) {
  for (int [HOLE]p ~~= 0; p < u; p++)~~
~~    e[p] +=~~ 1;
}
```

```
kernel void A(global int* e, int u) {
  for (int [HOLE] 1;
}
```

Figure 4: When a [HOLE] is inserted to a kernel at a random index, it hides a random number of tokens, unknown to BenchPress. On this example, BenchPress learns to predict the first hidden token, p.

3.3 Benchmark Generation

BenchPress's synthesizer operates as a generative model with the help of [HOLE] / [ENDHOLE] tokens. It receives an input with 1 or more [HOLE] tokens and returns a completed benchmark. For each [HOLE], BenchPress predicts one token that fits in the sequence at the [HOLE]'s index, with respect to its left and right context. If the predicted token is not [ENDHOLE], it moves the [HOLE] and all subsequent tokens one position to the right and inserts the predicted token to the initial target index. This intermediate kernel is iteratively provided as an input for the next token prediction and the process is repeated until BenchPress predicts [ENDHOLE]. This marks a [HOLE] is complete and the final sample is returned, as shown in Figure 5.



```
kernel void A(global int* e, int u) {
  for (int [HOLE] 1;
}
```

```
kernel void A(global int* e, int u) {
  for (int i [HOLE]1;
}
```

```
kernel void A(global int* e, int u) {
  for (int i = 0; i < u; i++)
    e[i] = abs(e[i]) + u;
  e[u] = [ENDHOLE]1;
}
```

```
kernel void A(global int* e, int u) {
  for (int i = 0; i < u; i++)
    e[i] = abs(e[i]) + u;
  e[u] = 1;
}
```

Figure 5: During sampling, BenchPress receives an input and predicts iteratively the fitting tokens. BenchPress predicts [ENDHOLE] to indicate a [HOLE] is complete.

On its own, this process only augments kernels given their existing left and right context. In that sense, BenchPress's language model is undirected with respect to the features that are targeted. We make BenchPress the first synthesizer to target desired parts of a feature space with beam search sampling. We generate a set of kernels from an empty input, we select the ones closer to the target features and we insert holes to generate new edited kernels iteratively.

Given a target feature vector, BenchPress samples a starting, fixed input feed `'kernel void [HOLE]'` and yields a collection of starting benchmarks. We reject benchmarks that do not compile and for the remaining we measure the Euclidean distance between their feature vectors and the target features. We select the *top-K* candidates that have the shortest distance from the target and we use them as inputs for the next generation. To improve diversity among promoted benchmarks we introduce randomness in the selection of *top-K* candidates: Each *top-K* sample, has a fixed probability $p = 0.15$ to be replaced by another random candidate of its generation. BenchPress lazily creates multiple different input instances for each selected candidate by placing a random [HOLE] of random length in order to synthesize a new sample. BenchPress generates a successive collection of benchmarks, of which K compiling ones with the shortest distance from the target again are selected with $\boldsymbol{p}$-randomness and used as inputs. This search continues until a sample achieves a distance of 0 from the target, or until a threshold of generations (i.e. beam search depth) is exhausted. BenchPress returns the closest benchmark to the target's features along with all beam search's intermediate benchmarks that cover the model's traversal of the feature space starting from the origin and ending near the target features. For the benchmark synthesis process, we use categorical sampling with temperature to sample BenchPress's probabilities. The sampling temperature, beam search's width K and depth are defined as sampling parameters.

In the worst case, BenchPress's directed program generation is slow, ranging from a few seconds to one hour, as it typically requires thousands of random language model inferences. However, BenchPress is the first program synthesizer that can target a set of desired program features. BenchDirect speeds up targeting features significantly as its directed language model requires far less samples per beam search iteration to produce samples close to the target features. Often, BenchDirect can target the feature space within one single inference step from an empty input.

3.4 Feature Space Search

A steerable synthesizer allows the generation of benchmarks with desired features. However, the automatic selection of those parts of the feature space that are worth targeting is challenging and depends on the downstream task.

BenchPress attempts to solve this by searching the feature space with query by committee [30], a well-known active learning technique. We implement a committee of (a) 7 NN, (b) 7 k-NN and (c) 7 K-means models. We set their initial state by passively training on a small portion of the downstream task's data. We sample the committee with thousands of random points in the space, we collect the predicted labels and measure the entropy for each sample. The entropy shows the level of uncertainty among the committee about the predicted label of a given point and is defined as:

$$H = -\sum^{l \epsilon L}(p(l) * \log(p(l))) \tag{1}$$

where L is the set of all predicted labels and $p(l)$ the probability of label l in the committee's prediction set for a given input. The highest entropy point is an important feature vector to target and BenchPress steers benchmark generation towards it with the approach explained in 3.3. We collect the labels of generated benchmarks and we train incrementally the committee with them. Then, we sample it to find the next highest entropy point. We continue this process until we saturate the feature space. BenchPress's committee is agnostic to the downstream task or the feature space and its I/O dimensions are hyper-parameters selected with respect to the task's feature and prediction dimensions.

3.5 Directed Language Modeling

BenchPress's synthesizer presented thus far is feature agnostic. This language model infills source code given the input context left and right of the [HOLE]. BenchPress is only able to steer program generation through a costly beam search on the model's output: we generate a large number of random code candidates and we feed those that are closer to the target features back into the model's input with new holes for further edits. Given BenchPress's language model is undirected, it often needs hundreds of thousands of code candidates to increase the chance of finding a few with the right features. This is inefficient and unsustainable on complex compiler tasks.

Instead of randomly trying to fill the space with new benchmarks to get closer to the target features, a more desirable approach to target them directly during synthesis is needed. Ideally, this would help generate a benchmark with the right features in a single inference. To this end, we develop BenchDirect, a steerable program generator that extends BenchPress's undirected language model into a directed one. Along with the masked source code input, BenchDirect also encodes its compiler features before masking. Its classification head selects tokens to fill a [HOLE] by jointly observing the code context and the encoded features. This leads to selecting tokens that are likely to generate a kernel that is (a) compiling (similarly to BenchPress) but also (b) matching the target features provided in the input.

BenchDirect's extended feature encoder is based on Transformer [37] and is shown in Figure 6. We encode a vector of numerical compiler features using an embedded layer with positional encoding followed by a Transformer-Encoder. We reduce the dimensions of the Transformer's output using a Fully Connected layer to match BERT language model's hidden state representation of its input source code. Both hidden states are concatenated and fed to a Fully Connected layer with GELU [20] activation to extract correlated features. Finally, a Decoding Fully Connected layer projects the joint hidden state into the vocabulary space. The feature encoder's input consists of 134 positions divided into three fixed segments. Each represents one feature space used in our evaluation: (a) 8 positions for Grewe's et al. features, (b) 56 for Autophase and (c) 70 for InstCount features. BenchDirect can support multiple spaces and it only needs to be trained once to direct benchmark synthesis on any of them. To steer generation in a new feature space, we simply need to extend a new segment in the Transformer-Encoder's input and apply fine-tuning using the new space's feature extractor to collect data from our training corpus.

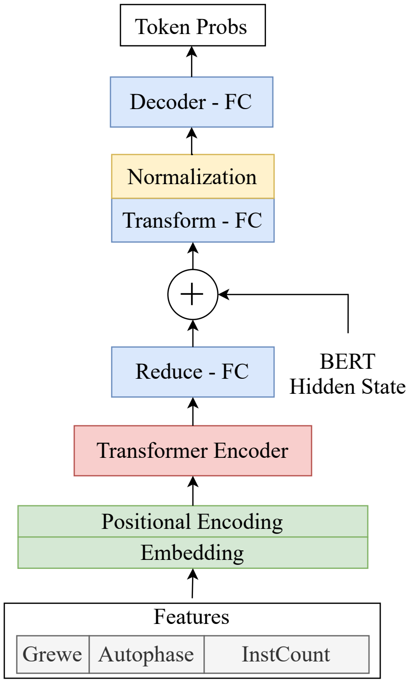


Figure 6: BenchDirect's directed language model design.

BenchDirect is trained with the same approach described in Subsection 3.2. We sample randomly one OpenCL kernel and introduce a [HOLE] to provide it to the language model's input. The model learns to predict the first token of the hidden sequence using cross categorical entropy loss function. Introducing compiler features in training is the distinction to this process. When one OpenCL kernel is sampled, its compiler features are also collected. The model receives a pair of inputs, (src_i, fv) and one output $token_i$, where i is the index at which the [HOLE] is located.

It is important to note that we do not feed the feature vectors of all three feature spaces to the encoder at the same time. Instead, we uniformly select one, we set its values to the respective segment of the encoder's input and we [PAD] all other positions such that gradients are not applied. Over training time, the model observes datapoints from all feature spaces for every kernel. Padding all feature spaces but one allows the trained model to learn how to direct synthesis to each one of them independently. Providing vectors from all spaces as one datapoint would possibly allow the model to learn correlations between them but this is not useful to us. What is more, directed synthesis on one of the feature spaces would be impossible. The model would have been trained to observe all three feature vectors for one given source code input. This means we would have to know the mapping function among all feature spaces to translate a target feature vector to all supported ones for the encoder's input. Instead, keeping one feature space per datapoint leads to the encoder's weights to be tuned accordingly to perform accurately on all spaces separately. Parts of the network (e.g. the FC layers) are jointly trained to optimise all feature spaces encoding. Other parts, such as the (Q, K, V) matrices are grouped in vectors, one for each index separately, and are only trained when their respective positions are not padded. An alternative solution would be to use many Transformer-Encoders, one per feature space, and train each separately. During generation, the appropriate Transformer would be manually selected given the desired feature space. Although this is a valid approach, there is no evidence to suggest it would perform better than one Transformer model large enough to learn all segments separately.

During sampling, BenchDirect receives a source code input and the target features as an input. Given the code context and the [HOLE] position, the model will attempt to select those tokens that will produce a compiling kernel with features as close as possible to the target in that respective feature space. At its best, we hope BenchDirect can receive an empty code input and provide the target benchmark at a single inference step. At the very least, the beam search sampler will go through fewer iterations and fewer inferences per generation compared to BenchPress.

4 EXPERIMENTAL SETUP

We describe the configurations used in training BenchPress, and the parameters used in evaluation, namely (1) Feature Spaces - we use three different representations of program features, (2) Target Benchmarks - We use Rodinia benchmarks [4] and their features as the target for synthesis by BenchPress, (3) Comparison to SOTA - we compare BenchPress with code synthesizers and human written code in improving Grewe's et al. heuristic model.

4.1 Platforms

We train BenchPress and conduct all our experiments on two 64-bit systems each having one Intel Xeon E5-2620 16-core CPU, 2x Nvidia GeForce GTX 1080 GPU and 32 Gigabytes of RAM. We use Ubuntu 18.04, PyTorch 1.9.1 [27], CUDA version 11.4 and Nvidia driver version 510.47.03. We use Clang-10 as BenchPress's compiler and LLVM-10 to compile and execute InstCount and Autophase [18] extracting tools. For compatibility reasons, we are required to use Clang LibTooling from LLVM-6 to execute Grewe's et al. [16] feature extractor.

4.2 Language Modeling for source code

We collect OpenCL code from GitHub and split it into single function instances. We ensure no kernels that come from benchmarks suites used in the evaluation are included in our corpus. We preprocess text, re-write variables and reject OpenCL kernels that do not compile. In total we mine 63,918 OpenCL kernels across 12,860 GitHub repositories and we successfully compile 19,637 of them (31% compilation rate).

We train BenchPress on our OpenCL Corpus for 10M steps with a batch size of 32. For BenchPress's BERT model parameters, we select 2 hidden layers, 12 attention heads. We set intermediate size, hidden size and max position embeddings to 768. We set the maximum length of holes to be 90% of a kernel's token length, i.e. a hole can hide almost all tokens of a training instance. We optimize the model using Adam optimizer with a learning rate that reaches a maximum of $45x10^{-6}$ after 20,000 warmup steps and decays linearly over the remaining training steps. We train BenchPress's language model to a final loss value of 0.28.

4.3 Feature Spaces

Compiler predictive models use static code features to represent programs and learn optimisation heuristics. A vector of independent characteristics represent a single program. Each of them are typically an integer or float value. Features are extracted at the Syntax level by traversing the AST or at the IR level using the compiler's middle end (e.g. LLVM-IR). A feature space is the collection of all possible program feature vectors.

BenchPress is a generative model that can be steered to generate samples for a desired part of the feature space. We evaluate BenchPress on three source feature representations we find across the literature, (a) Syntax-level Grewe's et al. features [16], (b) IR-level LLVM-InstCount [23] and (c) IR-level Autophase [18].

Grewe's et al. features are extracted with Clang's LibTooling and used to train their predictive model on the CPU vs GPU task for OpenCL kernels. This feature space holds 8 dimensions. 4 dimensions describe the number of 1) computational, 2) relational, 3) atomic and 4) memory access instructions. The feature space also counts the different type of memory instructions, local memory or coalesced. Finally, the computational to memory and coalesced to memory ratios are defined.

InstCount is a standard pass provided by LLVM-IR framework and used in Compiler Gym by Cummins et al. [6]. InstCount holds 70 dimensions: 67 dimensions each counting all 67 LLVM-IR instruction types and total number of 1) instructions, 2) basic blocks and 3) functions. Autophase by Huang et al. [18] holds 56 dimensions. While many of the features used in Autophase are shared with InstCount, they introduce new ones such as number of input arguments to PHI Nodes or total number of memory instructions. On the other hand, they do not include the count of some LLVM instructions that are not considered to contribute to a program's representation, e.g. CatchPad instruction.

4.4 Analysis of BenchPress and CLgen language models

CLgen [5] is the current state of the art in OpenCL benchmark generation. Its synthetic benchmarks improve the accuracy of Grewe's et al. predictive model [16] by 1.27×. However, Goens et al. [14] perform a case study and show evidence that CLgen's synthetic benchmarks do not improve the quality of training data and, consequently, performance of predictive models. They show that a predictive model in fact performs worse with synthetic benchmarks as opposed to human written benchmarks or code from GitHub.

This study motivates us to perform an analysis of BenchPress's language model, BERT, with CLgen in the task of undirected program generation. In this first experiment, we reproduce CLgen using the authors' artifacts and we sample it with a fixed input 'kernel void' to collect a dataset of unique OpenCL kernels. We use BenchPress on the same generative task and sample the model with the same fixed input 'kernel void [HOLE]' to obtain another dataset of unique benchmarks. In this experiment we focus on the language model's inference performance. We compare both generative models on their throughput, their ability to create compiling code, feature distribution and code size. In this experiment, we do not direct program generation. BenchPress generates compiling kernels in a single inference step.

4.5 Targeted Benchmark Generation

Next, we evaluate BenchPress's ability to steer towards desired program features. We use well-established compiler benchmarks as our reference and target their features within this space. These benchmarks usually perform intensive operations, such as matrix multiplications or FFT analysis, they contain hundreds of computational and memory instructions and are specifically fine-tuned by experts to exercise compilers from different angles. As a result, we believe features in these benchmarks provide a good target to assess performance of BenchPress's ability to target complex features.

We choose target benchmarks within the Rodinia suite [3, 4] as it is widely used in the literature [5, 7]. Similar to the training corpus, we collect the suite's source files, we inline header files and dependent OpenCL libraries into them, we split kernels into single source files and reject those that do not compile. In total, we collect 61 target Rodinia benchmarks out of which 58 compile. For the remaining benchmarks, we collect their features using the feature extractors for Grewe's et al., InstCount and Autophase feature spaces [16, 18, 23]. We target the feature vectors of these benchmarks and request BenchPress to generate at least one matching benchmark for each. We end up with three collective synthetic benchmark datasets, one for each feature space, that contain code with features matching Rodinia benchmarks. For each Rodinia benchmark's target feature vector, we measure the minimum Euclidean distance to it achieved between BenchPress, code from GitHub, CLgen and CLSmith [1, 39]. For GitHub's and CLSmith's kernels, we use SRCIROR [19] to apply code mutations exhaustively with beam search.

To make our experiment more intuitive we use two datasets for GitHub: a) GitHub consisting of all OpenCL kernels we collected and b) GitHub-768, a proper subset of GitHub which contains only the kernels that do not exceed BenchPress's sequence length of 768 tokens. Since BenchPress benchmarks' size are restricted to the architecture's sequence length, we feel it is important to make this distinction in order to present a view of BenchPress's actual performance on features that may be unreachable within the current sequence length. For example, it may be impossible to generate 2,000 computational instructions within 768 tokens. For such cases, we believe GitHub-768 with its equally restricted sequence length would allow for a fairer comparison.

For all three feature spaces, we weed out the Rodinia benchmarks that have an exact matching sample (i.e. a Euclidean distance of 0) in GitHub-768. Since we already have matching samples for them, we do not need to target them with BenchPress or any other generative model. However, we do not skip benchmarks whose features exist only in GitHub's full dataset as we wanted to explore the feasibility of using BenchPress to generate a sample with the same features but smaller sequence length. Applying this restriction we end up with 22 Rodinia benchmarks for Grewe's et al., 52 for InstCount and 36 for Autophase feature spaces.

We sample BenchPress for a maximum of 50 beam search iterations unless a benchmark matching the target features is produced. We set a workload size of 2048 samples per iteration. Among those of them that compile, our beam search sampler propagates to the next generation the closest 32 candidates, placing new holes into them.

4.6 Active Learning for Feature Selection

BenchPress's steerable generation is vital for searching the feature space while also finding useful features to target with active learning. In this experiment, we evaluate BenchPress in the downstream task of training the predictive model proposed by Grewe et al. [16], a well-tested problem used by many baseline models.

Grewe et al. train a decision tree model to predict the optimal device to execute a benchmark, choosing between a CPU and a GPU. They measure their model's performance as speedup achieved with using the predicted device for execution versus statically executing all benchmarks on the GPU. To train the predictive model, they use OpenCL benchmarks from 7 well-known benchmarks suites [5, 16]. In this experiment, we reproduce Grewe's et al. heuristic using their artifact and we also retrain it with datasets enriched with executable benchmarks from BenchPress using active learning and passive learning (i.e. targeting random parts of the feature space instead of searching it), CLgen and GitHub. We measure the speedup over static mapping for each of them.

To collect our evaluated datasets, we execute OpenCL benchmarks with CLDrive [5] by Cummins et al. CLDrive automatically generates inputs and drives kernels to the hardware. It measures the execution time per device across thousands of runs and it rejects kernels that produce runtime errors, do not modify any of the inputs (no output) or modify them differently for each run (not deterministic). For (a) the 7 human-written benchmarks suites, (b) BenchPress, (c) CLgen and (d) GitHub, we execute their kernel on CLDrive using a range of different *local* and *global size* configurations. We label each instance with the fastest measured device (the CPU or the GPU), in the same way Cummins et al. [5] and Grewe et al. [16] performed their evaluation.

4.7 Directed Language Modeling

BenchPress develops strong performance compared to state of the art program synthesizers and its benchmarks outperform even human-written benchmarks from GitHub in two tasks, (a) targeting the features of Rodinia benchmarks and (b) improving the accuracy of a compiler heuristic model. However, its undirected language model requires up to hundreds of thousands of inferences for its beam search sampler to minimize its samples' distance from the target features. This process can be inefficient, which we strive to address with a directed language model, namely BenchDirect.

We repeat the experiment of Section 4.5 to evaluate BenchDirect's accuracy and execution time in targeting the features of Rodinia benchmarks compared to BenchPress. We target the features of Rodinia benchmarks in all three feature spaces for a range of different workload sizes: 32, 64, 128, 256, 512, 1024 and 2048. A large workload size leads to a significant time overhead but is required to ensure high accuracy for BenchPress's undirected language model. This may not be the case for BenchDirect's directed synthesizer, speeding up directed generation without compensating on its accuracy. In this experiment, we explore how this parameter affects accuracy and total execution time for both models.

We re-train BenchPress and BenchDirect for 8M steps to a final loss of 0.14 using the same BERT hyper-parameters described in Section 4.2, except for their max position embeddings which we set to 512 instead of 768 to reduce training time. For BenchDirect's Transformer-Encoder, we set an embedding size of 512, 4 attention heads, 2 hidden layers and we set its Fully Connected layers to 1024 features. During sampling, we set the threshold of maximum beam search iterations to 5. Reducing the models' sequence length to 512 and the sampler's iteration threshold to 6 leads to a performance reduction compared to BenchPress's accuracy in Section 4.5. However, it saves valuable compute time. Both BenchPress and BenchDirect are restricted by this reduction, therefore the validity of this comparative study's results is not hurt.

4.8 Human Likeness of Generated Code

A great challenge for neural synthesizers is to produce programs that are human likely, that is following basic structural and syntactical form that makes them easy for humans to read and understand. The human likeness of a synthetic program reflects its quality and efficiency in the functionality it serves. To this end, we conduct a case study to measure the likeness of BenchPress's generated benchmarks to human-written code. We devise a double blind Turing test in which we show to human participants random samples from BenchPress, BenchDirect, CLgen, CLSmith and also human-written code from GitHub. They are shown randomly selected benchmarks from the stored datasets and are asked to label them as human or AI-written. We release our Turing test publicly available in the form of a web application[2].

5 RESULTS AND ANALYSIS

In this section, we show our experiments' results and compare BenchPress with state of the art techniques in OpenCL benchmark synthesis. We present case studies of (a) BenchPress's throughput as a generative model compared to CLgen, (b) its ability to steer benchmark generation towards desired features and (c) its performance in searching the feature space to enhance a downstream task's performance.

5.1 Analysis of BenchPress and CLgen language models

We perform an analysis of BenchPress and CLgen as language models and compare them in generating a collection of benchmarks from a fixed input feed, '`kernel void [HOLE]`' and '`kernel void`' respectively. We compare the two approaches measuring (a) the generative models' throughput and (b) the quality of their generated benchmarks in terms of code size and features. In this experiment, we do not use any directed search or iterative approach for BenchPress's generation. We perform this evaluation to measure how BERT, BenchPress's underlying language model, compares with CLgen as a generative model. Table 1 presents the aggregate measurements for the generated benchmarks using both approaches.

Compilation rate and code quality. BenchPress generates over 10× more unique compiling benchmarks than CLgen. This result is observed despite BenchPress generating 8× fewer unique benchmarks than CLgen. The compilation rate with BenchPress is 86% while CLgen has an exceedingly small rate of 2.3%. BenchPress's

[2] https://humanorai.co.uk

	# unique benchmarks	# compiling benchmarks	compilation rate	max tokens	max inst (LLVM-IR)	time per sample (ms)
BenchPress	190,460	142,607	86%	750	161	162
CLgen	1,564,011	13,035	2.33%	102	32	103

Table 1: Throughput comparison between BenchPress and CLgen on generated OpenCL benchmarks when BenchPress does not use feature-directed program generation.

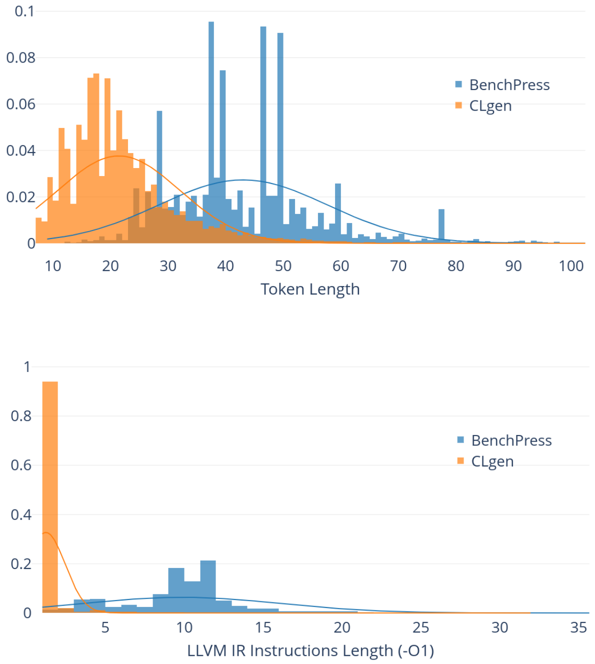


Figure 7: Probability distribution of (a) token length and (b) LLVM-IR Instruction count among BenchPress's and CLgen's generated benchmarks. BenchPress's benchmarks presented here are generated at a single inference step without iteratively directing program synthesis.

largest sample is 750 tokens compiling to 161 LLVM-IR instructions. This is a 7.5× and 5× increase in number of tokens and number of LLVM-IR instructions compared to CLgen's largest kernel. The only drawback of BenchPress compared to CLgen is that it is considerably slower in generating candidates. This is because the transformer-based architecture in BenchPress is significantly larger in number of parameters than CLgen's LSTM. Additionally, BenchPress tends to generate longer kernels than CLgen, necessitating more inference steps and longer generation time.

In Figures 7a and 7b, we show the frequency distribution of the number of tokens and number of LLVM-IR instructions for compiling kernels for both datasets. To visualize our results better, we focus on synthesized kernels with token lengths ≤ 100 and instructions lengths ≤ 25 where the vast majority of benchmarks are found. Most of BenchPress's benchmarks are found to have 20 to 80 tokens and 3 to 16 LLVM-IR instructions. The majority of CLgen's benchmarks are found to have 5 to 45 tokens and only up to 4 LLVM-IR instructions. 94% of CLgen's generated benchmarks have only 1 instruction when compiled to LLVM-IR. We analyze the dataset to explain this phenomenon and find CLgen generates a lot of comments, repeated dead statements and awkward non-human-like code such as multiple semi-colons. These results agree with the case study by Goens et al. [14] that shows the AST depth distribution of CLgen's code is significantly narrower compared to code from GitHub or standard benchmarks.

Feature space coverage. To further enhance our comparison, we perform an analysis on the feature space coverage of BenchPress's and CLgen's synthesized programs in all three feature spaces. Feature coverage is the most critical metric when evaluating the effectiveness of a benchmark synthesizer for predictive modeling. We use Principal Component Analysis (PCA-2) to represent the feature spaces in an easy to visualize 2-dimensional space. In Figures 8a, 8b and 8c we show the extent of feature space covered by candidates in the two approaches. CLgen's samples are clustered around the origin, while there is one outlier for Autophase and two for Grewe's et al. and InstCount features. Candidates generated by BenchPress are more scattered achieving a much wider coverage of the feature space.

5.2 Targeted Benchmark Generation

We use beam search to generate samples that target desired parts of the feature space. We compare BenchPress with human-written benchmarks from GitHub and synthetic benchmarks from CLgen and CLSmith in targeting the features of Rodinia benchmarks on three feature spaces. We use SRCIROR code mutator with beam search to collect GitHub and CLSmith benchmarks with closer features. For each target benchmark, we gather one OpenCL kernel per evaluated dataset whose features have the minimum available Euclidean distance from the target features. Figures 9a, 9b and 9c show the relative proximity of each benchmark to the target. This proximity is the complement of the relative distance of the two kernels, i.e, 1 minus the distance between the two kernels in the feature space relative to the distance of the Rodinia kernel from the axes origin. This allows us to express the quality of the match with an intuitive 0% to 100% scale: 100% means the two kernels have the same features, 0% means the best kernel is as close to the target as an empty kernel. We mark perfect matches with a white asterisk (*).

Performance on syntactic features. On Grewe's et al. feature space, BenchPress generates kernels that are the closest ones in features for all 22 Rodinia Benchmarks compared to CLgen and CLSmith, and 20 out of 22 compared to GitHub and GitHub-768. BenchPress synthesizes an exact match (100% relative proximity) for 14 target benchmarks.

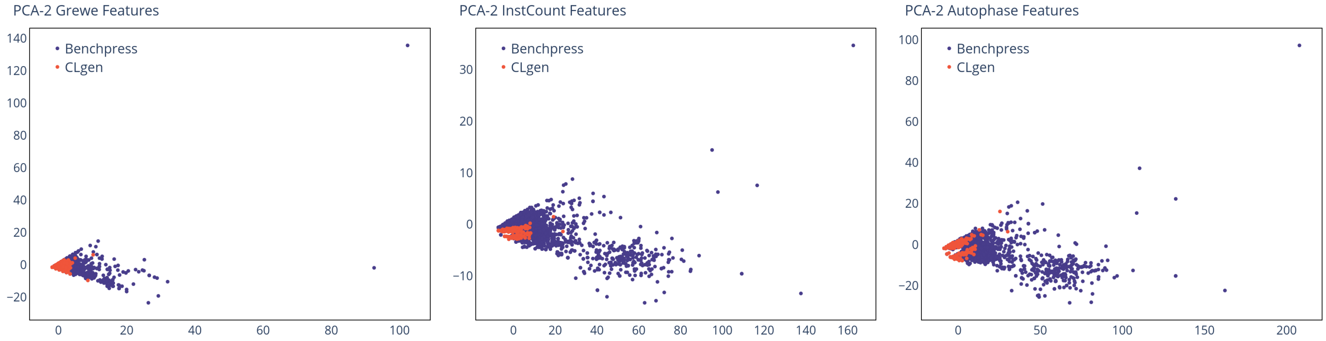


Figure 8: PCA-2 representation of feature space coverage of BenchPress and CLgen for (a) Grewe's et al., (b) InstCount and (c) Autophase feature spaces. In this experiment, BenchPress's generation is undirected and no iterative space search is performed.

We pick out and discuss a few examples from our results. The absolute distance achieved for 'nw-1' and 'ellipse_opt', is 1.0. For both targets, almost all features match except for one missing instruction (coalesced mem access and atomic inst respectively). For 'hotspot' GitHub and BenchPress produce a candidate kernel with exact matching features. However, BenchPress generates the matching candidate kernel in 421 tokens, unlike GitHub's closest benchmark that has 798 tokens. For the two target benchmarks that BenchPress's candidates were not closest to, we found only GitHub contains better samples for 'com_dwt-3' and and 'gpu-1', while BenchPress does not. We find both benchmarks to be fairly large (901 and 5,200 tokens respectively) and BenchPress cannot reach these features within 768 tokens. For the same reason, GitHub-768, CLgen and CLSmith does worse than BenchPress on these targets.

Performance on LLVM IR features. Autophase and InstCount features are extracted from the LLVM-IR of a program that has been compiled with -O1 flag to apply basic optimisations such as dead code elimination. BenchPress occasionally generates repeating operations that a compiler will remove or numerical operations that may be reduced to simple assignments. Owing to these optimisations, we find targeting benchmarks on these two feature spaces is more challenging than Grewe's et al. syntax-level features. With InstCount features, BenchPress generates candidates whose features completely match 2 out of the 52 Rodinia benchmarks. Among the remaining 50, BenchPress outperforms CLgen, CLSmith, GitHub and GitHub-768 for all target benchmarks, achieving higher proximity. SRCIROR significantly improves GitHub leading to GitHub+SRCIROR to achieve better proximity for 18 out of 52 Rodinia benchmarks compared to BenchPress. On Autophase features, BenchPress generates candidates matching the same 2 target benchmarks, while outperforming CLgen, CLSmith and GitHub on 30 out of 36 Rodinia benchmarks in total. GitHub+SRCIROR performs better than BenchPress for 8 out of 36 target benchmarks and produces an exact match for 'hotspotKernel'.

We previously explain the importance of having diverse features in compiler benchmarks and we show, in Figure 2, how sparse Rodinia benchmarks are on Grewe's et al. reduced feature space and how CLgen fails to provide any additional features. Now we introduce into this 2-dimensional space all BenchPress's kernels that are generated while performing directed space search to target Rodinia benchmarks and we present them in Figure 10. BenchPress densely populates the space around the target benchmarks that are clustered around the lower left corner. We find BenchPress's samples progressively converge to the target benchmark features with successive generations. For example, BenchPress targets 'com_dwt-3' at 385 computational and 137 memory instructions, starting from the axes origin and attempting to reach its features from different directions. One of the directions prevail but does not manage to exactly reach the target. The same happens for the top right point, 'gpu-1'. BenchPress's samples get closer developing a straight line from the origin to 1,000 computational and 100 memory instructions. At this point BenchPress is restricted by its sequence length and cannot augment further its samples. This is depicted by its attempt to reduce the distance by swapping the two instruction types within the same token length, forming a perpendicular line with a negative slope. We argue the area of Grewe's et al. feature space that BenchPress can cover within 768 tokens to be the area of the triangle formed by the intersections of the axes with the extension of the negative slope line developed by BenchPress's samples.

Summary - BenchPress vs GitHub vs CLgen vs CLSmith. 6 of the targeted Rodinia benchmarks exceed BenchPress's maximum sequence length of 768 tokens. In LLVM-IR feature spaces, care must be taken to generate code that will not be removed by compiler optimisations. This is a difficult challenge for source code generative models. However, our results demonstrate that BenchPress can generate OpenCL kernels that approach target human-written benchmarks compared to GitHub code and CLgen candidates. Our experiments also show BenchPress is dramatically better in all cases than CLgen, the current state of the art in OpenCL synthetic benchmark generation. We further elaborate on BenchPress's performance in the next subsections.

5.3 Active Learning for Feature Selection

We combine BenchPress's ability to generate benchmarks targeting desired features with active learning in order to generate benchmarks that improve the training of the Grewe et al. heuristic. We evaluate this against passive training with CLgen, GitHub code, and

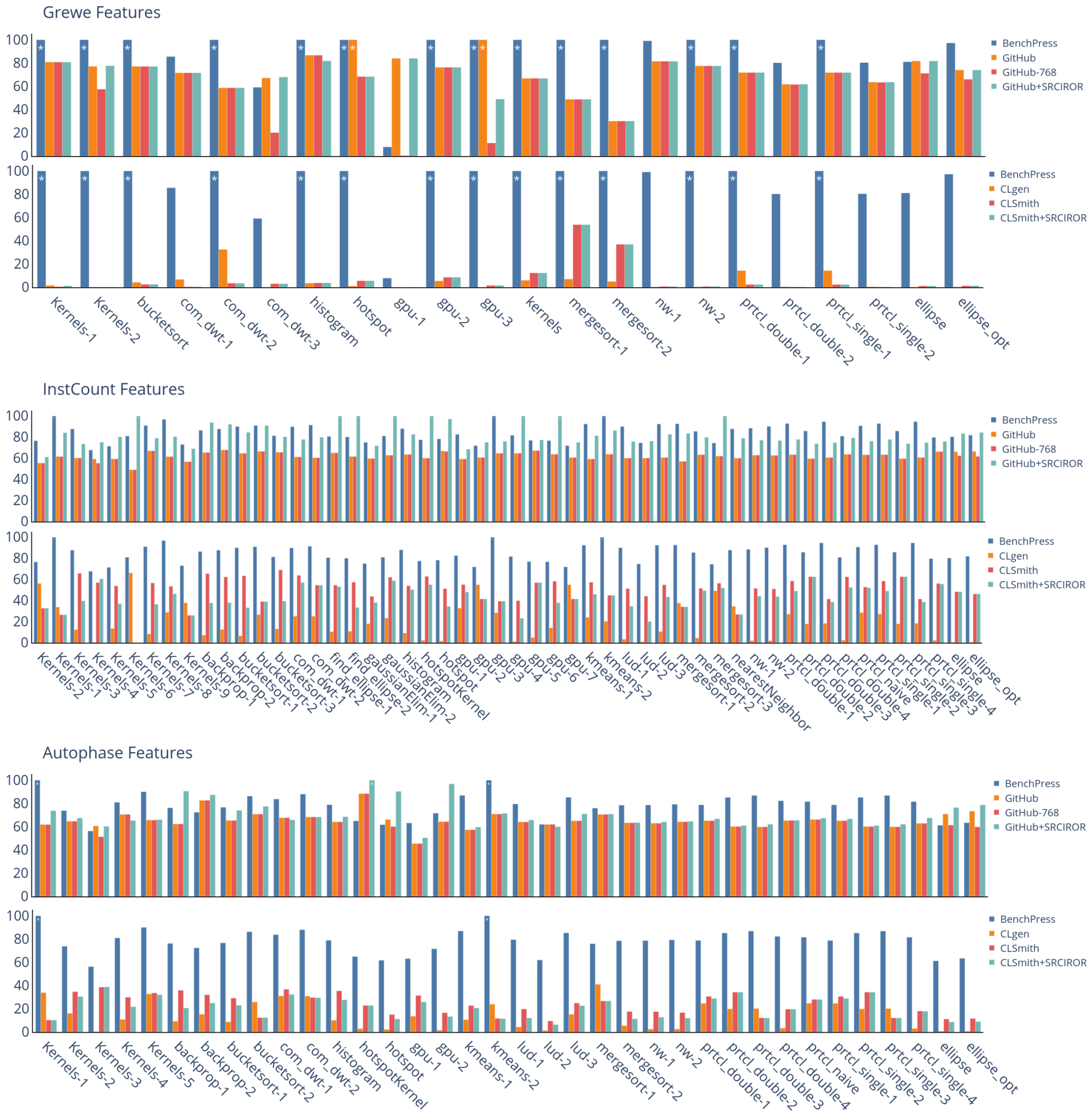


Figure 9: Relative proximity to each Rodinia benchmark of the candidate kernel with the closest features. We report the best match for seven datasets (`BenchPress`'s, `CLgen`'s, `GitHub`'s and `GitHub-768`'s datasets also combined with exhaustive mutations with `SRCIROR`) over three feature spaces ((a) Grewe's et al., (b) InstCount and (c) Autophase). Relative proximity is 1 minus the distance of the two kernels in the feature space relative to the distance of the Rodinia benchmark from the axes origin. 100% means an exact match in features and is highlighted with a white asterisk (*). A score towards 0% indicates the closest match is closer to the axes origin than the benchmark, i.e., a very small or empty kernel.

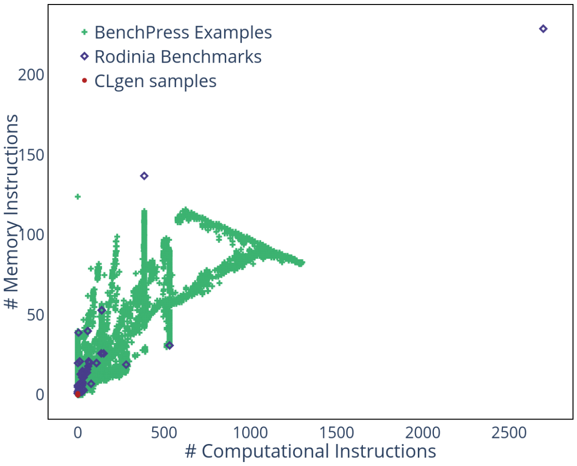


Figure 10: # Memory operations and # computational instructions for (a) Rodinia benchmarks in purple diamonds, (b) CLgen's samples in red dots and BenchPress's benchmarks in green crosses after performing directed search for all Rodinia benchmarks.

	Speedup %	Precision	Recall	Specificity
Benchmarks	+4%	0.81	0.86	0.61
BenchPress-AL	+6%	0.84	0.86	0.64
BenchPress-P	+1%	0.84	0.85	0.48
CLgen	-1%	0.52	0.86	0.43
GitHub	+1%	0.85	0.83	0.61

Table 2: Grewe et al. heuristic model's performance, precision, recall, and specificity when trained on each technique. Speedup is the geometrical mean of speedups over all benchmarks relative to the optimal static decision, i.e. running on the GPU. Precision, recall, and specificity treat GPU labels as positive and CPU labels as negative.

BenchPress with randomly selected target features. All approaches augment the same baseline training set that is taken from [5], containing 7 benchmark suites[3]. Table 2 shows the effect of each approach on the predictive power of the heuristic. Training only on human written benchmarks improves the heuristic's performance by 4%, as shown in Table 2's first row. To understand the maximum achievable improvement in the heuristic, we compute the best speedup (= 12%) that is achieved if the model chooses the optimal device as opposed to always picking the GPU. For 71% of the benchmarks, GPU is the optimal device, so no speedup improvement is possible. For the remaining 29% benchmarks, predicting the 'CPU' label correctly with Grewe et al. will result in a speedup improvement.

BenchPress using active learning (BenchPress-AL) clearly outperforms all other approaches in terms of average speedup, improving it by 6%. When trained on BenchPress with passive/random feature selection (BenchPress-P), the speedup achieved is only 1%. To our surprise, the same speedup is achieved with GitHub, which is worse compared with training only on the original benchmark suites. We further analyze the dataset collected from GitHub code and we find it to be imbalanced with 90% of its training instances are labelled as 'GPU'. This leads the model having a higher precision of 0.85, i.e. predicting correctly that a kernel should execute on the GPU, but falling short when it comes to correctly predicting the 'CPU' label. Training the heuristic with CLgen actually leads to a slowdown: it is 1% slower to execute kernels on the predicted devices compared to statically executing everything on the GPU, the baseline device. We analyze CLgen's dataset and observe the opposite pattern found in GitHub's dataset. 63% of its training data execute faster on the CPU than on the GPU. This is a direct consequence of CLgen generating small benchmarks that are poor in features, as the CPU may be slower than the GPU but the large overhead of transferring data to the GPU makes the CPU a better choice for small workloads. CLgen containing too many CPU-labeled kernel explains the heuristic's low precision and specificity, as it becomes biased to select the CPU very often leading to a slowdown.

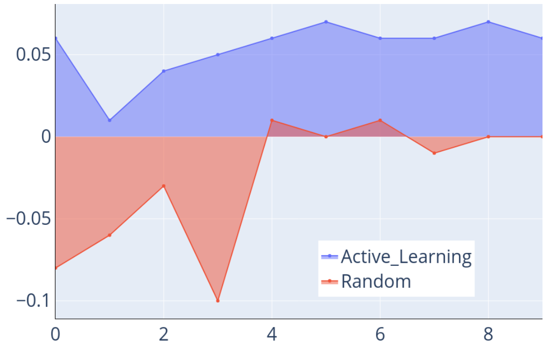


Figure 11: BenchPress's performance enhancement of Grewe et al. heuristic model when using active learning compared to passively targeting random parts of the feature space over the course of 10 sampling epochs.

Our main motivation behind using active learning is that it gives BenchPress the ability to target directly those parts of the feature space that will maximize a downstream task's performance. To assess the active learner's performance, we compare the Grewe et al. heuristic's speedup when trained on BenchPress's benchmarks that target areas of the feature space selected by the active learner versus benchmarks that target random features. In both cases, we execute BenchPress for the same amount of time, 10 sampling epochs (i.e., performing steered generation for 10 target feature vectors). In Figure 11, we show the speedup achieved by the heuristic when trained on the data collected at that step. Using active learning to target features, BenchPress's dataset improves the heuristic's speedup by 50% after 5 sampling steps, from 4% to 6%. Targeting random features never leads to a speedup higher than 1%. BenchPress can still develop the same speedup by targeting random features if infinite amount of time was available. Our active learner ensures that missing features are going to be quickly targeted, improving the state of the art within 5 sampling epochs.

[3] The benchmarks have been updated with a wider range of global and local sizes.

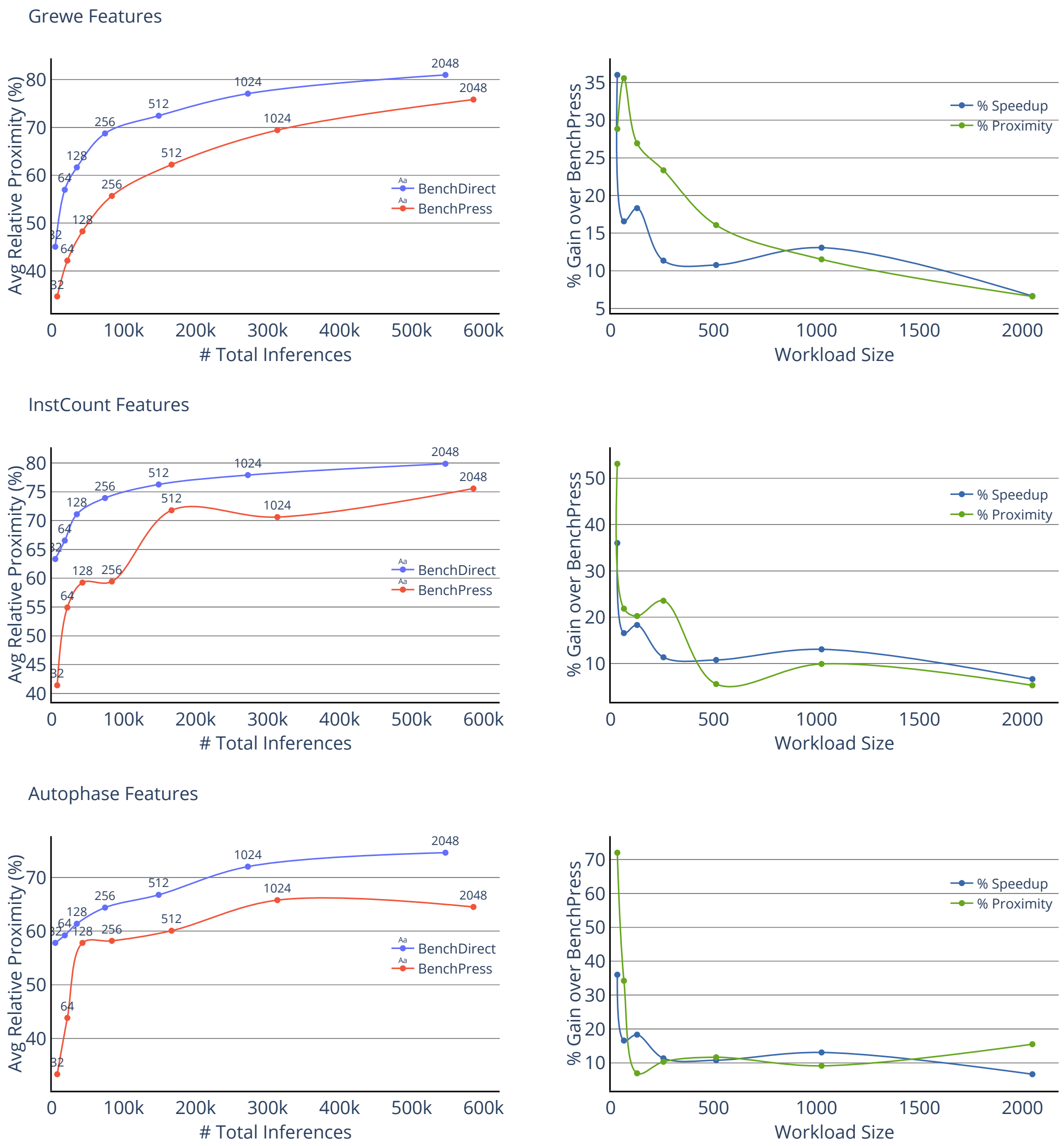


Figure 12: Pareto fronts of the average relative proximity versus total inferences in targeting Rodinia benchmarks over three feature spaces ((a) Grewe’s et al., (b) InstCount and (c) Autophase). Higher relative proximity and fewer inferences are better, therefore optimal points, i.e., Pareto-dominant, are those towards the top left. We annotate the workload size configuration per Pareto point. On the right, we show BenchDirect’s acquired speedup and accuracy gain over BenchPress for the same workload size setting.

5.4 Directed Language Modeling

We target the features of Rodinia benchmarks using BenchPress and BenchDirect. Both models use beam search over their synthesizer to minimize their samples’ distance from the target features. At the end of each search, we select the generated kernel whose features have the minimum Euclidean distance from the target benchmark. We perform this experiment for multiple beam search candidate sizes: 32, 64, 128, 256, 512, 1024 and 2048. On the left side of Figures 12a, 12b and 12c we show the Pareto fronts of the average relative proximity achieved over all Rodinia benchmarks versus the total amount of inferences. Relative proximity is defined in Section 5.2 as a percentage of how close a feature vector is to the target features relatively to the axis origins. Inferences are calculated as the number of beam search iterations to target all benchmarks multiplied by the workload size. Each datapoint is annotated with its workload size configuration. On the right side of Figures 12a, 12b and 12c, we show BenchDirect’s improvement in accuracy and execution time compared to BenchPress, for each workload size setting.

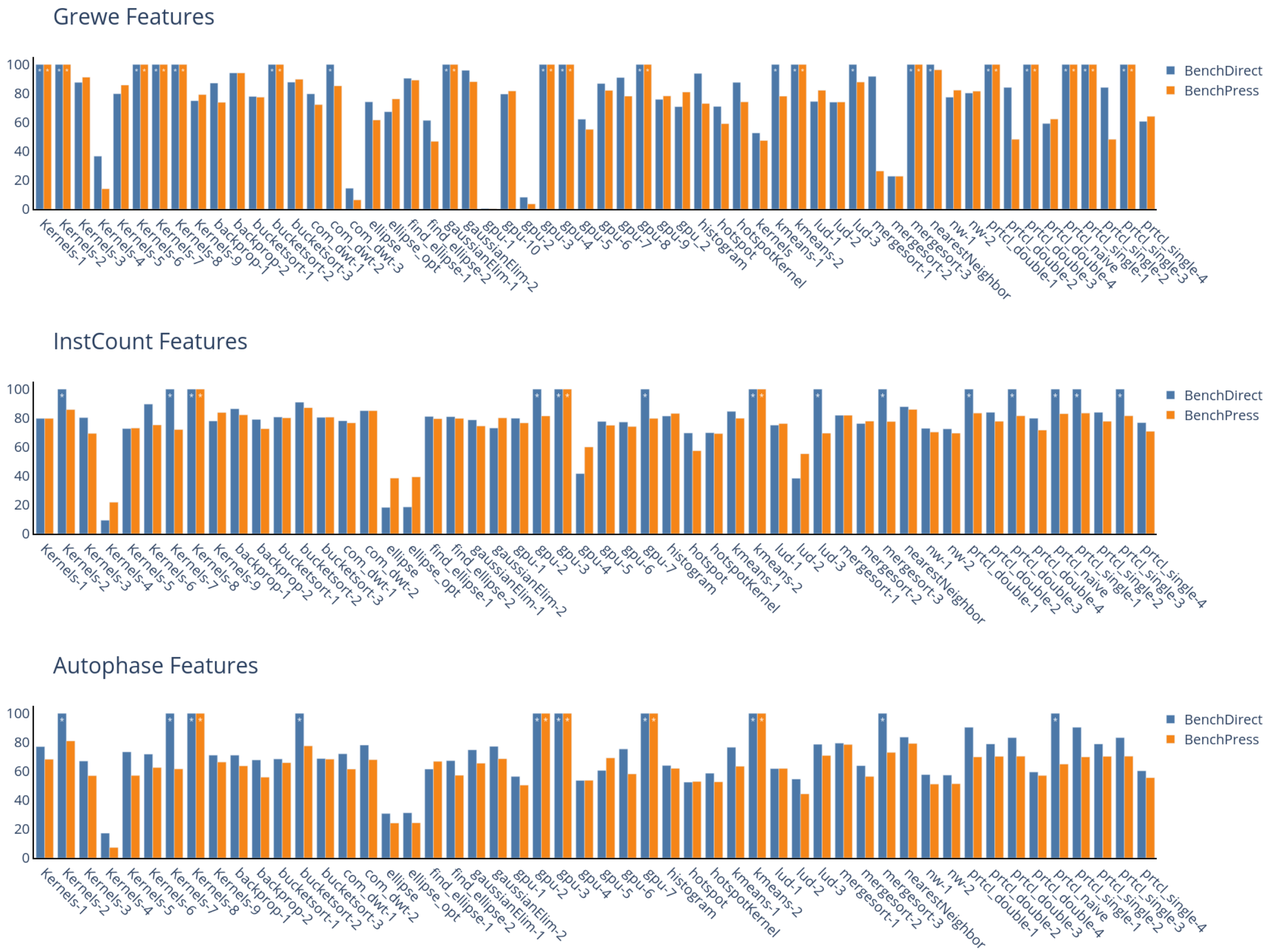


Figure 13: Relative proximity to each Rodinia benchmark of the candidate kernel with the closest features. We show the best match for `BenchDirect` and `BenchPress`. Relative proximity is defined in figure 9.

`BenchDirect` outperforms `BenchPress` in average relative proximity and total inferences for all workload size configurations, across all three feature spaces. Taking the average proximity and the execution time as a design space, the datapoints that are optimal with respect to these two metrics belong exclusively to `BenchDirect`, while there are no configurations for `BenchPress` that optimise either metric compared to the former. The effect `BenchDirect`'s directed language model has in targeting features is especially highlighted when the workload size is small. `BenchDirect`'s synthesizer conditions directly on the target features and provides, in very few attempts, candidates that match or are very close to them. This means a dramatic reduction in the amount of benchmarks per beam search does not drastically hamper the model's accuracy. The same is not true for `BenchPress`. While `BenchDirect` offers an average speedup of 10.2% and an improvement in average relative proximity of 10.1% for workloads greater or equal to 512, the speedup reaches up to 36% in all three feature spaces and the accuracy gain up to 72% on InstCount features for smaller workloads. This indicates `BenchDirect` remains consistent in the amount of iterations needed to achieve high accuracy, while `BenchPress` suffers in both areas.

Both models achieve a peak accuracy when they use a workload size of 2048. This is expected as generating more candidates increases the probability of getting closer to the target features. Using this configuration on both models, we show in Figures 13a, 13b and 13c the best relative proximity achieved for each target benchmark in all three feature spaces. Similarly to Figures 9a, 9b and 9c, candidates whose euclidean distance from the target is 0 (i.e., perfect match feature-wise) are marked with a white asterisk (*). For a selection of Rodinia target benchmarks, we show how the minimum distance from the target is reduced over the course of 6 beam search iterations for both models in Figure 14.

`BenchDirect` generates 1.8× more candidates that match exactly the target features compared to `BenchPress`. Specifically, it

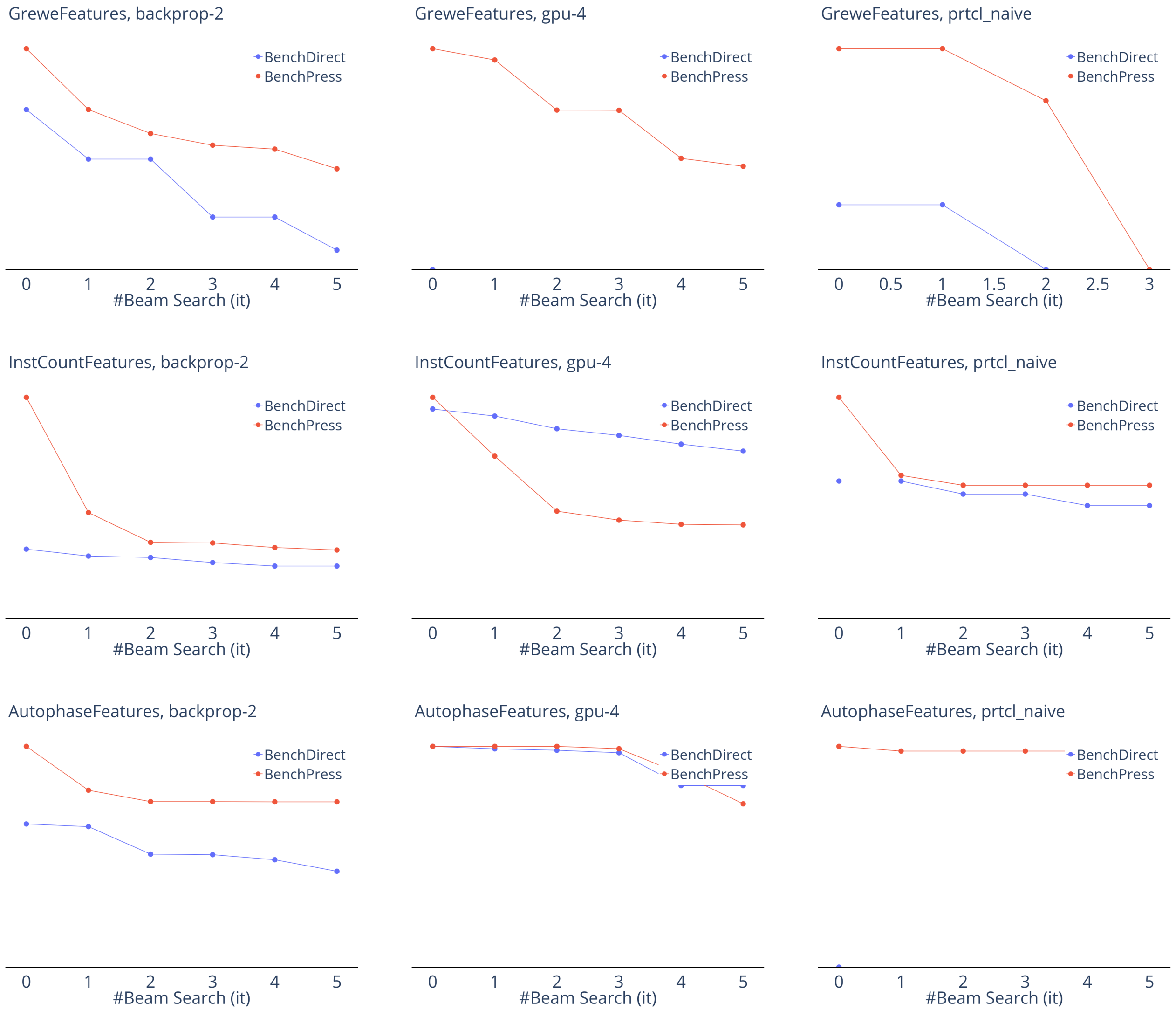


Figure 14: A qualitative comparison between BenchDirect and BenchPress for backprop-2, gpu-4 and particle_naive Rodinia benchmarks in all three feature spaces. We show for both language models the minimum distance achieved (y-axis) from the target over the course of six beam search iterations (x-axis).

matches 21 targets on Grewe's et al. features, 14 on InstCount and 10 targets on Autophase, compared to BenchPress's 17, 3 and 5 exact matches respectively. Overall, BenchDirect gets closer to the target compared to BenchPress. Its samples are closer, or as close, for 45 out of 58 Rodinia targets on Grewe's et al. features, 47 out of 52 on InstCount and 49 out of 52 on Autophase. BenchPress provides better candidates for 13, 5 and 3 targets on Grewe's et al., InstCount and Autophase features respectively. Even though it is expected for BenchDirect to miss some target features due to the experiment's randomness, we pick out a few such examples to discuss why this happens.

The largest performance gap in favour of BenchPress is observed on ellipse and ellipse_opt on InstCount features. These two benchmarks are very large, containing multiple thousands of instructions, therefore they are difficult kernels to target. We examine both models' generated samples over all 6 beam search iterations. In both cases, we find BenchDirect's closest candidate on the first iteration to be 8% closer to the target compared to BenchPress's. After measuring the distance distribution from the target for both models' samples, we find BenchDirect is 93% more likely to generate a sample whose distance is lower compared to BenchPress on the first beam search iteration. BenchDirect seems to succeed in these two target benchmarks indeed. However, at every inference step BenchDirect tries to match the target features in a single [HOLE] infill. As these two kernels are very large, this is a challenging task leading to most of its produced candidates to have syntactic errors, leaving it with only a few benchmarks that compile. Even though its first iteration's samples are closer

	Score %	#Human	#Total
GitHub	51%	139	270
BenchPress	53%	55	103
BenchDirect	49%	60	122
CLgen	38%	36	95
CLSmith	29%	26	89

Table 3: Score of 'human-likeness' expressed as the percentage of code examples from each dataset that were tagged as 'human-written' by users

compared to BenchPress, all successive iterations are becoming increasingly difficult for BenchDirect to produce a compiling kernel which also reduces the minimum distance. For that reason, BenchPress's random and cautious steps lead to benchmarks that are eventually closer. We notice this pattern to happen in all targets where BenchPress produced a better candidate. For these targets, it is likely that if we break down the difficulty into smaller steps by using intermediate feature vectors, this would have helped BenchDirect to get to the target features gradually but more accurately.

5.5 Human Likeness of Code

We conduct an empirical evaluation on BenchPress, BenchDirect, CLgen and CLSmith to measure the human-likeness of their samples by devising a Turing test in the form of a web application. Human-likeness is a desirable property for programs synthesized by generative models, as it indicates samples are likely to assimilate the functionality of human-written benchmarks. Each participant is shown a benchmark picked randomly from one of the 5 following datasets, (a) BenchPress, (b) BenchDirect, (c) CLgen, (d) CLSmith, and (e) GitHub. They are then asked to label the benchmark as written by a human or an AI. During this test, we show only the benchmarks that were selected in experiments 5.2 and 5.4, i.e., the closest samples per dataset to Rodinia for all 3 feature spaces. This results in 168 samples per presented dataset.

In total, we collect data from 77 participants that declare familiarity with programming. Table 3 shows how often users tag a test from each dataset as 'human-written'. We notice that human-written code from GitHub is classified as 'AI-written' by users in 49% of the tests. We believe this to be due to two reasons. First, the dataset from GitHub contains large OpenCL kernels that contain long and unnatural expressions or have had their loops manually unrolled for optimisation reasons, making them hundreds of lines long. Such kernels are most of the times labelled as 'AI-written'. Second, a participant may be suspicious of statements that do not look simple enough to be written by a human, therefore tending to select the 'AI-written' label more often.

Participants label samples from BenchPress as 'human-written' in 53% of its total tests and 49% of BenchDirect's total tests. While both scores are similar, it is likely that BenchDirect produces statements that are not likely written by a human slightly more often than BenchPress. This is because it tends to generate longer sequences than BenchPress when trying to reach to outliers of the feature space in a single inference step. CLgen's samples may look human likely but most of them are short, no longer than 3-4 lines. Often they contain no workloads or loops and are accompanied by unused arguments. This is the reason it scores lower at 38%. Finally, CLSmith is the most obvious case of unstructured and complicated code, being classified as 'human-written' only 29%. This fuzzer generates kernels by producing random expressions that conform to OpenCL's grammar, leading to random code whose functionality is not clear.

6 CONCLUSION

Predictive models for compilers have been shown to outperform compiler experts but they are restricted by the amount and quality of training data they are exposed to. What is needed is an approach that can synthesize benchmarks and enhance datasets with missing features. In this paper we propose BenchPress, a powerful code generator that uses active learning to search the feature space and steers generation towards desired features. BenchPress generates 10× more and 7.5× larger undirected benchmarks with 37× greater compilation rate than CLgen - a state of the art compiler benchmark generator - from a fixed input feed. BenchPress outperforms CLgen, CLSmith, code from GitHub and applied mutations with SRCIROR in generating OpenCL kernels that target the features of Rodinia benchmarks developed by human experts. BenchPress applies active learning to enhance Grewe's et al. dataset with benchmarks with missing features and leads to improving the heuristic's speedup by 50%. We further extend BenchPress's language model into a directed synthesizer given compiler features. This directed model produces 1.8× more matches to target features, it improves the generation process's accuracy by up to 36% and reduces inference time by up to 72%, while we show both our techniques outperform all other synthetic benchmark generation techniques in producing high-quality programs that are indistinguishable from human-written benchmarks. We hope this work to demonstrate a sustainable method to direct feature space search of program generation and that BenchPress's release to researchers will enable research in related domains.

7 RELATED WORK

BenchPress is inspired by BERT, a representation model by Devlin et al. [9]. Contrary to previous techniques [28, 29], BERT learns on unlabeled text data by jointly conditioning on both left and right context. BERT enables multiple applications of this architecture to a wide variety of difficult machine learning tasks, including programming languages. In CuBERT [22], Kanade et al. apply BERT over Python programs and evaluate it on finding typical mutation faults. In CodeBERT [11], Feng et al. fine-tune BERT to perform NL-PL and PL-NL transformations. In this work, we extend BERT to a bidirectional generative model, with the help of [HOLE] token.

Cummins et al. [5] develop CLgen, a deep learning generator based on LSTM [21] for OpenCL programs. They try to tackle the compiler benchmark shortage by providing synthetic benchmarks as training data for compiler heuristics. The authors present the Grewe et al. [16] heuristic model improved its performance by 1.27× when trained on their synthetic benchmarks. However, Goens et al. [14] show that training with CLgen's synthetic samples lead to a slowdown compared to training on human-written benchmarks only. To explain this, they measure the AST depth of

CLgen’s samples and show it is 3× smaller compared to human-written benchmarks and code from GitHub and poor in features, therefore unrealistic. This motivates us to develop BenchPress, which produces 10× more unique kernels that are 7.5× larger on average.

In 2019, Nye et al. develop SketchAdapt [26], which uses a generator-synthesizer [2, 10] to generate program sketches given I/O specifications. The synthesizer samples sketches and the generator fills <HOLE> tokens with statements. SketchAdapt performs better than other architectures [2, 10], however it samples only a pre-defined pool of operations, which restricts its diversity. Bruen et al. [8], propose a Tree2Tree approach for code generation using VAE. They encode AST nodes using Tree-LSTMs (Tai et al. [33]) and train their model on C++ functions. They test their approach against a VAE with an LSTM Seq2Seq model. They use their model as a synthesizer by sampling random AST representations which they extend to new programs. Their Seq2Seq model achieves a compilation rate of up to 67% with greedy search, however this happens because the model greedily selects the most probable labels, leading to repetitive samples. When sampling with temperature, their Tree2Tree architecture is able to generate a wider variety of samples, but only achieves a compilation rate of 22%, which translates to a few functions.

Gupta et al. [17] develop SED, a two-stage generator. A synthesizer receives I/O specifications and generates programs likely to satisfy them and a neural debugger applies program repair to reform them into functions that match specifications. Gupta et al. evaluate three synthesizer architectures and measure (a) the correctness of generated programs across tests and (b) the accuracy of their debugger to repair code. While SED is an innovative work, Karel is a small-scale language and SED’s generative performance on a complex programming language is not evaluated. Faustino et al. develop Anghabench [7] to tackle the benchmark shortage [5, 38]. Anghabench is a collection of C programs mined from GitHub. In order to make it compilable, they use Psyche-C [25] type inference engine to apply type reconstruction and resolve dependencies. Structs, unions and other composite data types are omitted or re-declared with primitive types. Their benchmarks are compiling, but cannot be executed. Compared to AnghaBench, BenchPress resolves type dependencies of composite types and user-defined functions without changing the functionality or semantics of programs.

REFERENCES

[1] [n.d.]. https://github.com/ChrisLidbury/CLSmith. [Online; accessed 25-Apr-2022].

[2] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. (11 2016).

[3] Rodinia Benchmarks. [n.d.]. http://lava.cs.virginia.edu/Rodinia/download.htm. [Online; accessed 25-Apr-2022].

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[5] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 86–99. https://doi.org/10.1109/CGO.2017.7863731

[6] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. arXiv:2109.08267 [cs.PL]

[7] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 2021. ANGHABENCH: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 378–390. https://doi.org/10.1109/CGO51591.2021.9370322

[8] Sander de Bruin, Vadim Liventsev, and Milan Petković. 2021. Autoencoders as Tools for Program Synthesis. arXiv:2108.07129 [cs.AI]

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*.

[10] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 990–998.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[12] fivosts. [n.d.]. https://github.com/fivosts/BenchPress.git. [Online; accessed 1-Sept-2022].

[13] GitHub. [n.d.]. https://docs.github.com/en/rest. [Online; accessed 25-Apr-2022].

[14] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. 2019. A Case Study on Machine Learning for Synthesizing Benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 38–46. https://doi.org/10.1145/3315508.3329976

[15] Google. [n.d.]. https://cloud.google.com/bigquery. [Online; accessed 25-Apr-2022].

[16] Dominik Grewe, Zheng Wang, and Michael F. P. O'Boyle. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–10. https://doi.org/10.1109/CGO.2013.6494993

[17] Kavi Gupta, Peter Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis.

[18] Ameer Haj-Ali, Qijing (Jenny) Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. 2020. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 70–81. https://proceedings.mlsys.org/paper/2020/file/4e732ced3463d06de0ca9a15b6153677-Paper.pdf

[19] Farah Hariri and August Shi. 2018. SRCIROR: A Toolset for Mutation Testing of C Source Code and LLVM Intermediate Representation. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 860–863. https://doi.org/10.1145/3238147.3240482

[20] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian Error Linear Units (GELUs). https://doi.org/10.48550/ARXIV.1606.08415

[21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (nov 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[22] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. arXiv:2001.00059 [cs.SE]

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*. San Jose, CA, USA, 75–88.

[24] Yann LeCun, Y. Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521 (05 2015), 436–44. https://doi.org/10.1038/nature14539

[25] Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. 2020. Type Inference for C: Applications to the Static Analysis of Incomplete Programs. *ACM Trans. Program. Lang. Syst.* 42, 3, Article 15 (nov 2020), 71 pages. https://doi.org/10.1145/3421472

[26] Maxwell Nye, Luke B. Hewitt, Joshua B. Tenenbaum, and Armando Solar-Lezama. 2019. Learning to Infer Program Sketches. In *ICML*.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[28] Matthew Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. (02 2018).

[29] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training.

[30] H. S. Seung, M. Opper, and H. Sompolinsky. 1992. Query by Committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory* (Pittsburgh, Pennsylvania, USA) *(COLT '92)*. Association for Computing Machinery, New York, NY, USA, 287–294. https://doi.org/10.1145/130385.130417

[31] OpenCL specification. [n.d.]. https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html. [Online; accessed 25-Apr-2022].

[32] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[33] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1556–1566. https://doi.org/10.3115/v1/P15-1150

[34] Foivos Tsimpourlas, Lazaros Papadopoulos, Anastasios Bartsokas, and Dimitrios Soudris. 2018. A Design Space Exploration Framework for Convolutional Neural Networks Implemented on Edge Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2212–2221. https://doi.org/10.1109/TCAD.2018.2857280

[35] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. 2023. BenchPress: A Deep Active Benchmark Generator. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/3559009.3569644

[36] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. 2021. Supervised Learning over Test Executions as a Test Oracle. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1521–1531. https://doi.org/10.1145/3412841.3442027

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[38] Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532