A Case Study on Machine Learning for Synthesizing Benchmarks

Andrés Goens
TU Dresden, Germany
andres.goens@tu-dresden.de

Alexander Brauckmann
TU Dresden, Germany
alexander.brauckmann@tu-dresden.de

Sebastian Ertel
TU Dresden, Germany
sebastian.ertel@tu-dresden.de

Chris Cummins
University of Edinburgh, UK
c.cummins@inf.ed.ac.uk

Hugh Leather
University of Edinburgh, UK
hleather@inf.ed.ac.uk

Jeronimo Castrillon
TU Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Abstract

Good benchmarks are hard to find because they require a substantial effort to keep them representative for the constantly changing challenges of a particular field. Synthetic benchmarks are a common approach to deal with this, and methods from machine learning are natural candidates for synthetic benchmark generation. In this paper we investigate the usefulness of machine learning in the prominent CLgen benchmark generator. We re-evaluate CLgen by comparing the benchmarks generated by the model with the raw data used to train it. This re-evaluation indicates that, for the use case considered, machine learning did not yield additional benefit over a simpler method using the raw data. We investigate the reasons for this and provide further insights into the challenges the problem could pose for potential future generators.

CCS Concepts • **Computing methodologies → Machine learning**; • **Software and its engineering → Automatic programming**.

Keywords Machine Learning, Benchmarking, Synthetic program generation, CLGen, Generative models

ACM Reference Format:
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 (MAPL '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3315508.3329976

MAPL '19, June 22, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6719-6/19/06…$15.00
https://doi.org/10.1145/3315508.3329976

1 Introduction

Benchmark programs are instrumental for the evolution of new compiler technology, hardware architectures and runtime systems. Established benchmark suites cannot be up to date and cover every domain, which is why many approaches resort to synthetic benchmarks [3, 7, 8, 13]. The last years have seen developments in machine learning (ML) that have proven to be very successful in tasks like image processing, natural language processing [28] and have even had some success in tasks related to programming languages and source code [1]. These methods from ML are an evident candidate to tackle the problem of synthetic benchmark generation.

Recently, Cummins et al. [5] proposed to use a generative model of code based on LSTM [9] to produce synthetic benchmarks. With a framework for mining code samples from open source repositories in the Github platform and a driver to produce pertinent input data, they gathered thousands of kernels to train an LSTM model and produce code that looked and behaved like the human-written kernels. This is an innovative use of machine learning for generating synthetic benchmarks, and to the best of our knowledge, no other similar approaches have been proposed. However, while the authors assessed their synthetic benchmarks as means to enhance traditional benchmark suites, they did not show that a generative model was necessary in this case. In particular, the authors missed a comparison between these synthetic kernels and the original mined kernels they used to train the generative model.

In this paper we use the published artifacts of [5] to re-evaluate CLgen (Section 2). Apart from comparing the performance of the different benchmark sets for tuning a heuristic, as done in the CLgen paper, we look into the datasets themselves to gain more insight into the nature of the problem (Section 3). Our results show that the dataset derived from the mined github kernels yields better results than the synthetic one. This casts a doubt on whether the use of machine learning, concretely the generative model in CLgen, was

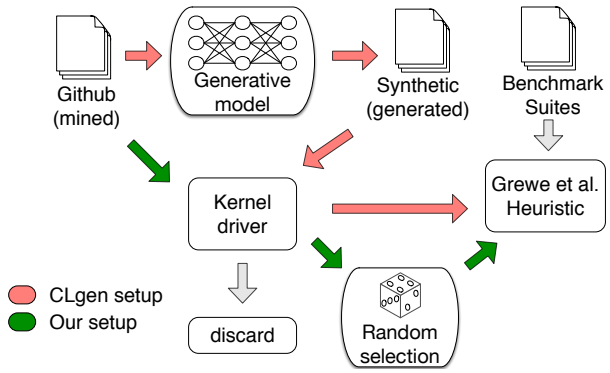

Figure 1. The experimental setup used in [5] and our changes.

useful in this particular case. We discuss the results and pose hypotheses of what could be the reason for them, in terms of the heuristic being trained, the objective for assessing the performance of the heuristic, and the generative model of code (Section 4). We also analyze use cases from recent papers in prominent conferences to confirm how most uses of benchmarking in the programming languages and compiler domain follow similar goals as those from CLgen.

2 Case study: CLGen

In this section we describe CLgen [5] and re-evaluate its use of machine learning, specifically generative models.

In heterogeneous systems, the problem of deciding where to execute a computation is far from trivial, even in the single CPU-GPU case. Independent of the solution approach, to test how good a solution is we need a representative benchmark set. A common solution strategy is to tune heuristics [4, 27], which require rich benchmarking as well.

In [5], the authors used OpenCL kernels mined from Github and used them to train a generative model based on a character-level representation of code, using a neural network based on the LSTM architecture [9]. Kernels must compile and require input data to execute, which is needed to evaluate performance. To execute a generated kernel, a kernel driver was designed which rejects kernels if it cannot compile and generate data for them to run. The authors use the kernels synthesized by this generative model to enhance the benchmarks used to train the heuristic from [27], training the heuristic with both, the benchmarks and their synthetic kernels. Figure 1 shows a high-level view of this process.

2.1 Experimental Evaluation

To test the usefulness of generative models on the CLgen use case, we compared the original setup used by [5] with an equivalent setup using the training data set mined from Github, as shown in Figure 1. We first compare how well both data sets, the training set (Github) and the benchmarks synthesized by CLgen (Synthesized), fare at the task they were needed for, i.e., training the heuristic presented in [27]. Since the hand-written benchmarks can cover many cases that will make the difference between these two data sets, including them in the training set will impede the comparison of the two sets. Thus, we isolate the effects of both code sets by removing the hand-written benchmarks from the setup and testing the heuristic only trained with the github/synthesized data sets. We also replicate the full experiment from [5], which includes the benchmark suites enhanced with synthetic kernels, and compare that to the benchmark set enhanced with a random selection of the Github kernels instead.

Table 1. Experimental Environment

	Intel CPU	NVIDIA GPU
Model	Core i7-7700K	GeForce GTX 1080 Ti
Frequency	4.20 GHz	1544 MHz
No. of cores	4	3584
Memory	16 GB	11 GB

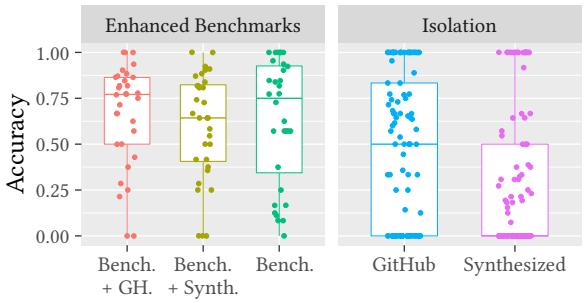

Figure 2. The accuracy of the heuristic with different setups. On top of the box-plots, the actual points are overlayed with a (random) horizontal jitter for improving the visualization.

As in [5], we divide the benchmarks into the same (disjoint) training and test sets. We measure their execution using two different compute devices, further described in Table 1. For the synthetic data set we use the same 1000 kernels produced in [5], as provided in the corresponding artifact. For the Github data set we used a random selection of the same size, 1000 kernels, out of the $\approx$ 1500 mined and automatically executable Github kernels. Since this includes randomness in the selection of the kernels from the Github set, we repeated the experiments 100 times and report the results of the experiment run with the median accuracy obtained with this random process. This is also important for further analysis of the data, since we can analyze the selection of kernels that yielded the median results.

Figure 2 shows a box plot of the accuracy obtained by the heuristic, trained with the different data sets, both the Github and synthetic sets in isolation (without the benchmarks), as

well as the benchmark set enhanced with each of the two, and the benchmark set alone. The accuracy is calculated by comparing how many times the heuristic correctly identifies the right device for execution of a kernel, compared to the ground truth, i.e., on which device it actually executes faster. We see that, in isolation, the heuristic is unmistakably trained better with the original training set than with the kernels synthesized by the generative model. In fact, as can be seen, the median accuracy of the model trained only with synthetic kernels is 0%. This means that for more than half the data points from the test set it got the prediction wrong. Thus, in isolation, the data set is a better benchmark set than the synthesized benchmarks, at least as measured by how well it trains the heuristic from [27] as tested with the benchmarks used in [5].

The intended use case for CLgen, however, is to enhance the benchmark suites, not to replace them. To asses this we can compare the results of the enhanced benchmark set in Figure 2. It shows that using the kernels mined from Github we also obtain better results than with the benchmarks synthesized by the generative model. It shows an even more interesting phenomenon, however: By enhancing the data set with the synthesized kernels, the heuristic actually got *worse*. We will discuss this further in Sections 3 and 4.

While for the learning aspect the accuracy with which the right device is predicted is interesting, it is not the most relevant aspect for a practical assessment of the heuristic. It is much more important for the heuristic to predict the right device when the difference in execution times is significant between CPU and GPU, whereas for kernels with negligible differences, making the right prediction is not as important. For this, we can take a look at the overall speedup of selecting the application with the heuristic vs executing all kernels in the device that is fastest for most of them. Figure 3 shows a comparison of the overall speedup of the two setups enhancing the benchmarks with the synthetic and GitHub kernels, respectively. Each bar shows the average speedup for all data points in that benchmark. The figure shows that for many benchmarks this setup brings an overall slowdown. This is also confirmed by the average speedup overall, or slowdown rather. The set enhanced with synthetic benchmarks fares worse than with the github set, in line with the results from the accuracy analysis.

Finally, we discuss one advantage that the generative model has over the Github set, as well as the hand-crafted benchmarks, namely that the generative model can produce an arbitrary amount of kernels. Indeed, while the kernels produced by the generative model should all be similar, it is plausible that if given the chance to produce enough kernels, the model can generalize and produce kernels with more interesting feature vectors that enrich the generated set. To test the effect of the number of kernels produced, we varied the number of kernels given to enhance the benchmark set in the same setup as above (see Figure 1). The results of this can be seen in Figure 4. The tendency is pretty clear, and statistically significant, since for more than 50 additional samples, the hinges do not overlap [26]. It shows that using the Github set as training data seems to produce better results than the synthetic set with any number of samples. More importantly however, the figure shows how the speedup worsens when adding more kernels, which indicates that the advantage of being able to produce an arbitrary amount of kernels does not provide much benefit in this concrete use case. It is important to note that the heuristic has only a handful of parameters and it probably will not benefit from more training data as much as other, more complex models would. This probably explains why more data lead to worse results. This issue will be discussed further in the following sections. It is important to mention at this stage, however, as it the results from experiment from Figure 4 might be very different with more complex models, effectively limiting the generality of this result.

3 Analysis of the Data Sets

In this section we investigate the data points used in the last section. The goal of this section is to understand some of the limitations of the approach and gain insight into the problem itself.

We consider the feature spaces of the different data sets as defined by the features from [27]. For this, we calculate the principal components as for the join of all three sets, namely Github, Synthesized and the original benchmarks. Figure 5 depicts the ground truth for the decision space, evaluated for all the points from all considered sets. The decision considered is the one being made by the heuristic: whether to execute a kernel in the CPU or the GPU. Even with the logarithmic scale it is clear that several points with the same coordinates have different values. What this shows is that it is a poor model to consider the decision as a smooth and regular function GPU(x). Not only is it very irregular (and actually discrete), it is not even a function because it has different values on the same coordinates. This also holds true in the full four-dimensional space, which is hard to visualize in a plot.

Obviously this is an artifact of the features considered: they are not sufficient to decide whether to map a computation to a CPU or a GPU. However, it also serves to explain why more training data does not necessarily improve the heuristic. This provides a model that could explain what happens in Figure 4, where the additional synthetic examples actually worsen the heuristic for the benchmarks used for testing. Since the space is irregular and not easily dividable into decision regions, some of the additional points could be serving as noise and skew the heuristic in a direction that makes it worse for the code we are interested in improving.

Consider Figure 6, which shows the relative frequencies of the (joint) first principal component, i.e. with the highest

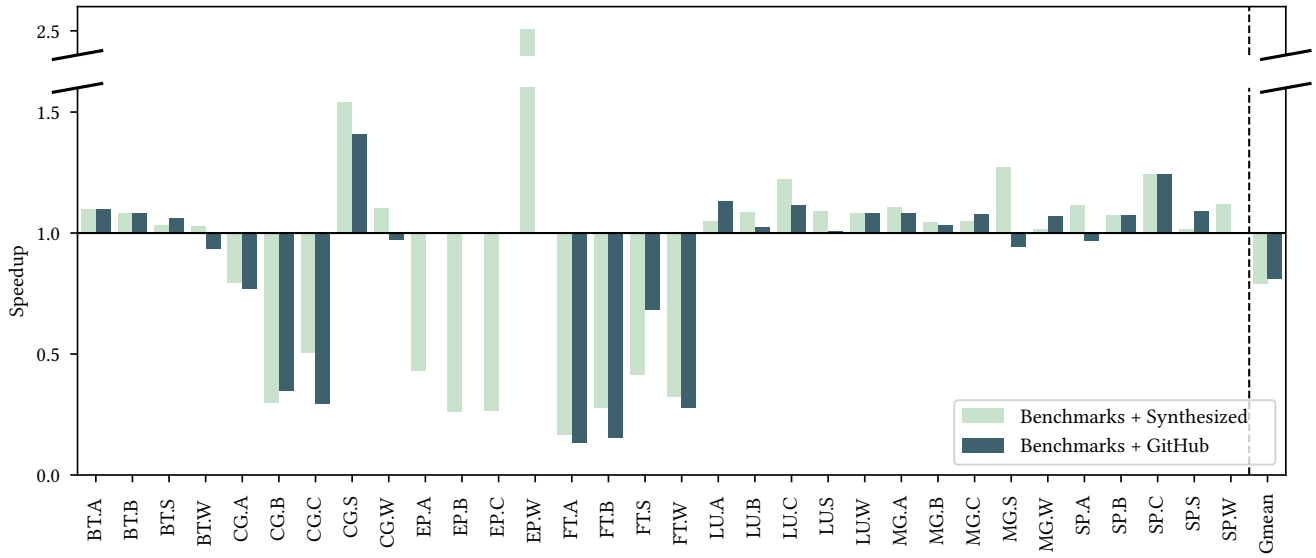

Figure 3. The speedup obtained with the trained heuristic for the individual benchmarks.

Eigenvalue, for the three data sets. Instead of a histogram, it shows the kernel density estimate, which is a smoothen version of the histogram, making the visualization clearer. While this is merely a one-dimensional projection onto a relevant linear combination of the features, it serves to make an intuitive assessment of the relation between the distributions of the data sets. We see how the benchmark set has code with features not present in the Github set, whereas the Synthesized set seems to be a proper subset of the Github set, in terms of this principal component.

A final interesting insight from analyzing the data can be seen in Figure 7. It shows again a histogram of the relative

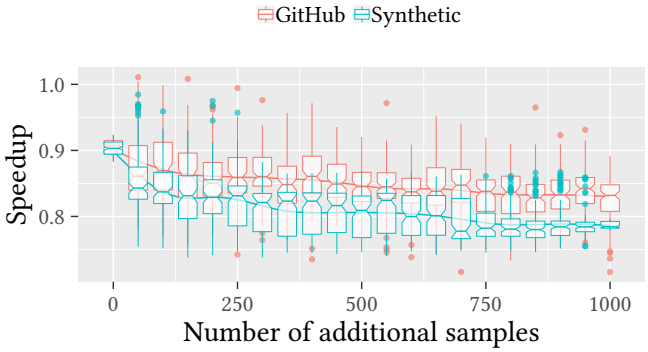

Figure 4. The average speedup as function of the number of kernels selected from the set. The plot shows notched box-plots (see [26]) for 100 repetitions with different random selections for each measured number of samples.

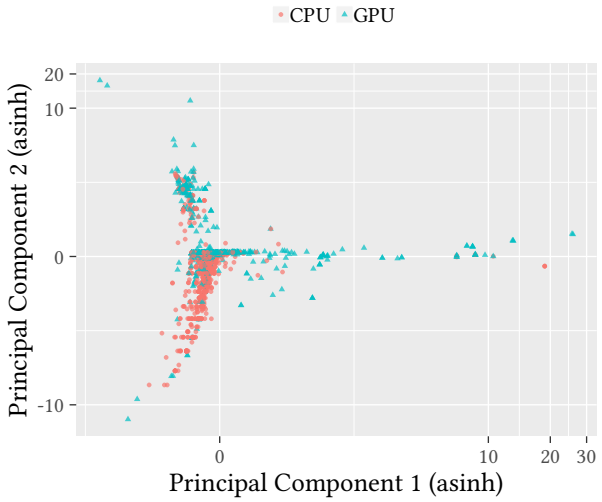

Figure 5. The ideal mapping for all points considered.

frequencies of kernels in all three data sets. This time, however, instead of considering the principal component, we characterize them by the depth of the abstract syntax tree (AST). We use this as a metric to approximate the overall size and complexity of the kernel. It is interesting to note that the benchmark suite contained significantly more kernels with a deeper AST. This opens up an interesting discussion about the usefulness of benchmark suites as a whole, which we will briefly retake in Section 4. However, an in-depth discussion is a matter of future work.

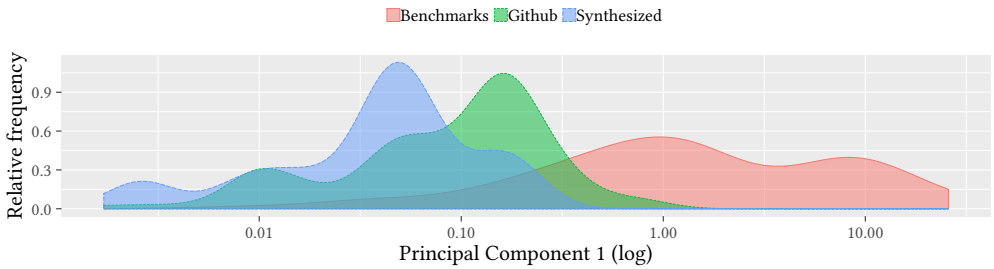

Figure 6. Smoothened estimations of density with respect to Principal Component 1 of the code spaces of all three data sets.

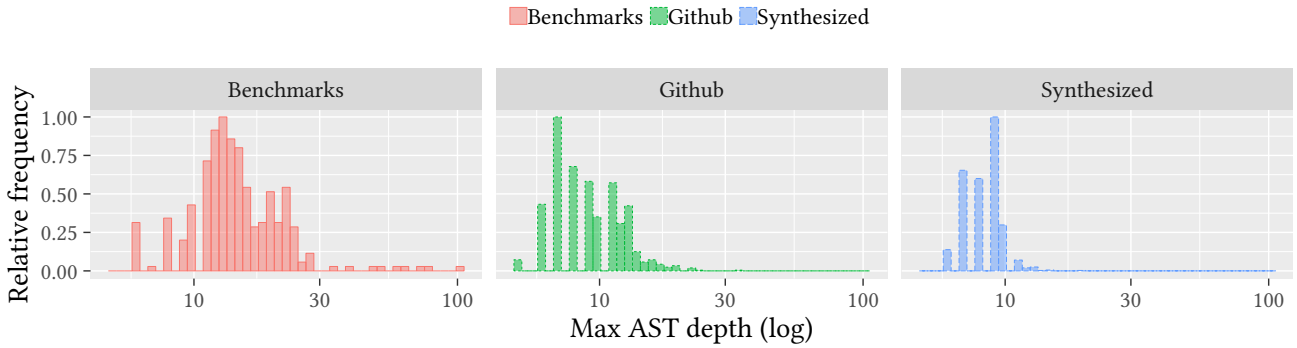

Figure 7. Relative frequencies of code by maximal AST depth.

Similarly interesting in Figure 7 is that the synthesized code tends to be shorter than the original Github programs. We will also discuss this further in Section 4.

4 Discussion

With our experimental evaluation and analysis of the datasets, we have pointed at issues with the state-of-the-art generative model presented in CLgen for benchmark generation, as it pertains to training the heuristic from Grewe et al. [27]. Since several aspects play a role, we can only partially speculate the specific reasons for the different effects seen in the previous sections. In this section we will consider three concrete aspects, namely the Grewe et al. *heuristic*, the *objective* how the goodness of the heuristic is measured, and the *generative model* itself.

4.1 The Heuristic

About the heuristic we can draw a clear conclusion from Figure 5. As it was pointed out, this space is not only very irregular, but we cannot even model the decision where to place a computation as a mathematical function in the feature space. This is because there are points that share the same feature vectors, where the decision of placing onto the CPU or GPU is different. Similarly, the lack of clustered regions in the decision space shows that the features considered are insufficient for separating the space properly into regions, which is necessary for the heuristic to work well.

An analysis as the one we presented from Figure 5 can be very useful to asses the goodness of manually designed features. For this, having a generator that can produce a virtually unlimited number of data points could be very useful for exploring the space, especially if conditional generation can be used to steer the generation towards particular points in the feature space. However, it is important to point out a potential caveat with this. All points produced by a generator with the same feature vector can have the same value for some function (e.g. the CPU or GPU decision). Even then, it does not mean that no examples exist where the value of the function is different but the feature vector identical. These values can just be extremely unlikely to be produced by the generator.

Finally, many issues found with manually-designed feature spaces like the one from the Grewe et al. heuristic are solved by using methods from deep learning [18]. Indeed, for this particular problem and some related ones, a deep-learning based approach was shown to outperform the heuristic [4].

4.2 The Objective

In Figure 2, it seems that both the Synthesized and Github sets, on their own, perform poorly as a training set for the

heuristic. As explained with Figure 5, this likely results from a poor coverage of the feature set, as indicated by the principal component analysis. However, we can take a different perspective on this fact. The reason why the heuristic trained with these two sets performs poorly is that the sets (of feature vectors) differ significantly from the benchmarks used to asses the goodness of the trained heuristic. However, we could question if the problem lies in the latter instead, namely using these benchmarks used to asses the goodness of the trained heuristic.

Consider the following scenario. An open source project, say the Freedesktop project, knows it has several OpenCL kernels as part of their project and wants to optimize it. The developers know of a compiler with an optimization that can estimate where to best execute their kernels by analyzing static features of the source code (the heuristic from this case study) and decide to give it a try. In what case will the compiler yield the best results for them, if it was trained using the standard benchmarks, or if it was trained used the mined training set from github? Clearly, not everyone using an OpenCL optimization is an open source project developer publishing to Github, but we argue that this scenario is still worth a closer look. To this end, we repeat the setup investigated in the experiments of Section 2 (see Figure 1) to compare the Github dataset with the Benchmarks. This time, however, we exclude all mined kernels from the Freedesktop project from the training set (concretely, 91 OpenCL kernels) and use them as test set instead of using the benchmarks.

A summary of the results of this can bee seen in Table 2. It shows that indeed, the model trained with the benchmark set performs worse than with the Github set when evaluated on their performance on sources from an actual (open-source) project, instead of on other benchmarks. This is a sign that there might be issues with using traditional benchmark suites. Mining source code from online repositories could be a viable alternative. Note however, that the benchmark suites performed better when tested on the mined set than the mined set on its own when tested on benchmarks. This is especially clear when comparing the speedups in Table 2, where the average speedup is just slightly higher (6%) for the heuristic trained with the kernels mined from Github than with the traditional benchmarks. A systematic investigation, far beyond our setup here, would be required to understand these issues well.

Table 2. The Grewe et al. heuristic as tested on the Freedesktop kernels

Dataset	Accuracy (avg)	Speedup (avg)
Github	0.73	1.06
Benchmarks	0.48	1.00

Besides the potential issue with traditional benchmarks, the experiment above sheds light on another point. Benchmarking plays two distinct roles in this case study, and the usefulness of the different datasets for those two roles seems to differ as well. On the one hand, the benchmarks are used as **input** data for tuning the heuristic proposed by Grewe et al. On the other hand, the benchmarks are also used to **characterize** the improvement yielded by this heuristic. It is easier to asses how good a set of benchmarks performs as input to the heuristic than it does as a test set to characterize the performance. This is clearly the case because how good a set of benchmarks is for characterizing something like performance depends on the use case and its objectives, i.e, what is being characterized. In particular, to asses how good a benchmark is for the characterization of a property we first need to answer what that means precisely. In the following we propose a concrete formalization of what we mean for a benchmark to be good at characterizing a property.

Let $\mathcal{P}$ be a property of code, like the speedup obtained from using the heuristic. There is an implicit probability density function p over all possible programs, describing the probability of the code to be used as input to the compiler using the heuristic[1]. To test the speedup, we want to calculate the expected value $E[\mathcal{P}]$ over this implicit pdf p. For testing, we argue that we want a *representative* benchmark. Ideally, we would get a set of programs $x_1, \ldots, x_l \sim p$ i.i.d., where p is the implicit probability density function of code been written. The expected value $E[\mathcal{P}]$ can thus be approximated arbitrarily well with growing sample size l. We do this because, in the example, we assume that the users of the compiler will also draw from this distribution p, and thus E[speedup] tells us what speedup the users can expect to get out of the optimization.

We believe the distinction between the usage of benchmarks is important, and researchers could benefit from a differentiated approach to the usage of benchmarking, depending on how the benchmarks are being used. This is especially true in ML-enabled methods, which are bound to increase the amount of benchmarks being used as input. Many people are already looking at these two problems in a differentiated matter. To show how this is the case, we classified papers using GPU benchmarks from two prominent conferences, Code Generation and Optimization (CGO) and Parallel Architectures and Compilation Techniques (PACT), between 2013 and 2016 that used GPU benchmarks. Table 3 shows a table summarizing the results from this classification. The table shows, in particular, that the scenario of characterizing a property is indeed a very common use case for benchmarks. Incidentally, for all papers that did not fit these two scenarios, i.e. classified as "Other", could be grouped

[1] A compelling case can be made that in some cases it is the "dynamic" property of the probability that a piece of code will be *executed*, not necessarily written or used as input to the compiler, that is most interesting here.

Table 3. Analysis of 20 research papers from 2013–2016 CGO/PACT papers and their respective use of benchmarks in their evaluation. The column "Arch" refers to the type of architecture used, and the column "Type" gives a synopsis of what the benchmarks were used for. The column "Classification" tries to classify the usage of the benchmarks in the two described above, as an Input (e.g. to train or tune some heuristic), or to characterize a property. The properties being characterized are listed under the "Metrics" column.

Paper	Arch	Type	Benchmarks	Metrics	Classification
Margiolas et al. (accelOS) [25]	GPU	compiler, runtime system	Parboil	Fairness Improvement Throughput Speedup	Characterization
Anantpur et al. (VTW) [2]	GPU	warp scheduler	Rodinia Parboil	Mean Performance Improvement	Characterization
Wu et al. (gpucc) [31]	GPGPU	OpenCL compiler for CUDA	Rodinia, SHOC, Tensor	Compilation Speedup Runtime Speedup	Characterization
Kim et al. [17]	CPU	code analysis for locality-centric scheduling	Rodinia Parboil	Runtime Speedup Cache Misses	Characterization
Li et al. [22]	GPGPU	compiler-driven automatic data placement	Rodinia CUDA SDK	Performance Speedup Optimality	Input
Fauzia et al. [6]	GPU	uncoalesced data access optimization	Rodinia Polybench	Performance Speedup Optimality	Characterization
Margiolas et al. [24]	GPGPU	portable host-device communication opt.	Rodinia Parboil	Performance Speedup Allocation Overhead	Characterization Other
Wang et al. [29]	GPGPU	analysis of hybrid memory	Rodinia CUDA SDK	Normalized Energy, Miss Rates, IPC Avg Access Latency	Other
Kayiran et al. (DYNCTA) [15]	GPGPU	optimization for thread-level parallelism	Rodinia CUDA SDK Parboil	IPC Improvement Core Utilization Improvement Fetch Latency Improvement	Characterization
Lee et al. (SKMD) [20]	CPU/GPU	CPU-GPU collaboration	AMD SDK NVIDIA SDK	Performance Speedup	Characterization
Jia et al. (Starchart) [12]	GPU	statistical auto-tuning	Rodinia NVIDIA SDK	Prediction Accuracy Power/Performance Trade-off	Input Other
Ji et al. (RSVM) [11]	CPU/GPU	software virtual memory for CPU-GPU	AMD SDK NVIDIA SDK	Problem Size Cap Algorithm Performance	Characterization
Kaleem et al. [14]	CPU/GPU	adaptive CPU-GPU scheduling	TBB, Parsec Rodinia	Performance Speedup	Characterization
Jablin et.al. [10]	GPU	adaptive trace-scheduling for GPUs	Rodinia	Performance Speedup	Characterization
Lee et al. (CAWS) [21]	GPU	criticality-aware scheduling	Rodinia Parboil	Performance Speedup Prediction Accuracy	Characterization
Xu et al. (PATS) [32]	GPGPU	branch divergence-aware warp scheduler	Rodinia Parboil, ISPASS	Common Divergence Patterns Performance Impact	Other Characterization
Lee et al. (VAST) [19]	GPU	automatic GPU virtual memory management	NVIDIA SDK	Performance Speedup Page Lookup Overhead	Characterization
Magni et al. [23]	GPU	ML-based thread coarsening	AMD SDK NVIDIA SDK Parboil	Speedup Distribution Coarsening optimality likelihood Performance Speedup	Other Characterization
Wang et al. (OAWS) [30]	GPU	memory occlusion-aware warp scheduler	Rodinia, SHOC PolyBench, Mars	Normalized IPC Performance Load instruction percentage	Characterization Other
Kim et al. [16]	GPGPU	Automatic pipeline-parallelism for dependent kernels	NVIDIA SDK Rodinia, Parboil, PolyBench	Performance Speedup Sensitivity Study	Characterization Other

into a third scenario. Instead of obtaining the expected value of a property $E[\mathcal{P}]$ the papers are interested in a characterization of this property $\mathcal{P}$ over different points in the space of possible programs. In most cases this served to guide the works towards cases where the property being investigated was particularly interesting and concentrate on those.

4.3 The Generative Model

Finally, the results seen in this paper are probably also in part due to the concrete generative model itself. For example, the generator seems to be biased towards generating shorter kernels, as was discussed and can be seen in Figure 7. In particular, this effect seems to change the distribution in comparison to the Github kernels, and fixing it could plausibly improve the results.

Little can be said in general since, to the best of our knowledge, no other benchmark generators based on machine-learning exist that we could compare to. More and different models should be proposed and compared to CLgen if we want to improve generative models of code and understand their limitations. For example, as CLgen is based on LSTM, it understands and represents code by nature as a sequence (in this case of characters). It produces a large amount of invalid kernels, that will not compile, which could in part be attributed to this. An argument can be made that the structure of code is best described as a graph, as graphs capture logical dependencies better than one-dimensional representations from sequences. Proposing a generator using a graph-based representation on code and comparing it to CLgen would provide insight into the effects of this to code generation, if any.

5 Conclusions

In this paper we have re-evaluated the use of machine learning in CLgen, a method for mining OpenCL kernels from Github and training a generative model from them to produce OpenCL benchmarks. In our re-evaluation we considered the question, for the use case from the original CLgen paper, if using the generative model for synthesizing benchmarks provided advantages over the data used to train said generative model. The in-depth analysis in this re-evaluation indicates that for this concrete use case, the generative model was not useful. The synthesized benchmarks produced results that were consistently similar to, and slightly worse than, those produced from using the mined kernels instead.

An analysis of the heuristic that used the benchmarks for tuning, as well as its problem space, show that a significant amount of limitations can be attributed to the heuristic itself. Similarly, the generative model used, or the hand-crafted benchmarks used for testing the heuristic present issues of their own. Each of these issues requires additional attention in order to understand its role in the results we found in this case study, and should be addressed in future work. We believe that the analysis presented in this paper may help other researchers inform their decisions when considering benchmarks to test their own hypotheses.

Acknowledgments

We thank Gil Lederman and the rest of the participants of the Spring 2018 CS 294-147 seminar at UC Berkeley for valuable discussions on this subject. This work was supported in part by the Center for Advancing Electronics Dresden (cfaed), the German Academic Exchange Service (DAAD) and the Studienstiftung des deutschen Volkes.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

[2] J. Anantpur and R. Govindarajan. Taming warp divergence. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 50–60, Piscataway, NJ, USA, 2017. IEEE Press.

[3] A. Chiu, J. Garvey, and T. S. Abdelrahman. A language and preprocessor for user-controlled generation of synthetic programs. *Scientific Programming*, 2017, 2017.

[4] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[5] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather. Synthesizing benchmarks for predictive modeling. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages 86–99. IEEE, 2017.

[6] N. Fauzia, L.-N. Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on gpus. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 12–22, Washington, DC, USA, 2015. IEEE Computer Society.

[7] A. Goens, S. Ertel, J. Adam, and J. Castrillon. Level graphs: Generating benchmarks for concurrency optimizations in compilers. In *Proceedings of the 11th International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG'2018), co-located with 13th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, Jan. 2018.

[8] T. D. Han and T. S. Abdelrahman. Use of synthetic benchmarks for machine-learning-based performance auto-tuning. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1350–1361. IEEE, 2017.

[9] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] J. A. Jablin, T. B. Jablin, O. Mutlu, and M. Herlihy. Warp-aware trace scheduling for gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 163–174, New York, NY, USA, 2014. ACM.

[11] F. Ji, H. Lin, and X. Ma. Rsvm: A region-based software virtual memory for gpu. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 269–278, Piscataway, NJ, USA, 2013. IEEE Press.

[12] W. Jia, K. A. Shaw, and M. Martonosi. Starchart: Hardware and software optimization using recursive partitioning regression trees. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 257–268, Piscataway, NJ, USA, 2013. IEEE Press.

[13] A. Joshi, L. Eeckhout, and L. John. The return of synthetic benchmarks. In *2008 SPEC Benchmark Workshop*, pages 1–11, 2008.

[14] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA, 2014. ACM.

[15] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the*

22Nd International Conference on Parallel Architectures and Compilation Techniques, PACT '13, pages 157–166, Piscataway, NJ, USA, 2013. IEEE Press.

[16] G. Kim, J. Jeong, J. Kim, and M. Stephenson. Automatically exploiting implicit pipeline parallelism from multiple dependent kernels for gpus. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 341–352, New York, NY, USA, 2016. ACM.

[17] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu. Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 257–268, Washington, DC, USA, 2015. IEEE Computer Society.

[18] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[19] J. Lee, M. Samadi, and S. Mahlke. Vast: The illusion of a large memory space for gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 443–454, New York, NY, USA, 2014. ACM.

[20] J. Lee, M. Samadi, Y. Park, and S. Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[21] S.-Y. Lee and C.-J. Wu. Caws: Criticality-aware warp scheduling for gpgpu workloads. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 175–186, New York, NY, USA, 2014. ACM.

[22] C. Li, Y. Yang, Z. Lin, and H. Zhou. Automatic data placement into gpu on-chip memory resources. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 23–33, Washington, DC, USA, 2015. IEEE Computer Society.

[23] A. Magni, C. Dubach, and M. O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 455–466, New York, NY, USA, 2014. ACM.

[24] C. Margiolas and M. F. P. O'Boyle. Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 55:55–55:65, New York, NY, USA, 2014. ACM.

[25] C. Margiolas and M. F. P. O'Boyle. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 82–93, New York, NY, USA, 2016. ACM.

[26] R. McGill, J. W. Tukey, and W. A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[27] M. F. O'Boyle, Z. Wang, and D. Grewe. 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)*, pages 1–10. IEEE Computer Society, 2013.

[28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. Technical report, Technical report, OpenAi, 2019.

[29] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao, and J. S. Vetter. Exploring hybrid memory for gpu energy efficiency through software-hardware co-design. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 93–102, Piscataway, NJ, USA, 2013. IEEE Press.

[30] B. Wang, Y. Zhu, and W. Yu. Oaws: Memory occlusion aware warp scheduling. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 45–55, New York, NY, USA, 2016. ACM.

[31] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. Gpucc: An open-source gpgpu compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 105–116, New York, NY, USA, 2016. ACM.

[32] Q. Xu and M. Annavaram. Pats: Pattern aware scheduling and power gating for gpgpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 225–236, New York, NY, USA, 2014. ACM.