Learning-Based Hardware Acceleration Speedup Estimation

Minwoo Kang *
University of California, Berkeley
mkang@eecs.berkeley.edu

Nikolay Velkov
University of California, Berkeley
nikolayvelkov@berkeley.edu

Abstract

On-chip power and area constraints require SoC designers to compare the merits of synthesizing different programs as accelerators. However, building an analytical model to estimate speedups from acceleration can be challenging because modern SoCs contain complex out-of-order CPUs. On the other hand, full system simulations are also too slow to be integrated into an iterative SoC design space search algorithm. In this paper, we present the first learned performance model that classifies a program by its expected speedup from high-level synthesis (HLS) hardware acceleration. We show that even with relatively light-weight learning models, such as random forests and gradient boosting, our classifier implementations can achieve up to 76.7% accuracy and on randomized C programs.

1 Introduction

With the slowdown of Moore's law and Dennard scaling, a growing number of modern System-on-a-Chip (SoC) architectures are including specialized accelerators as part of their designs. Research over the past years have confirmed the efficacy of accelerators, especifically how they can achieve orders of magnitude improvements in computation latency, throughput, and power consumption over general-purpose processors such as CPUs [24]. As a result, accelerators have gained significant interest and have been developed to target a number of application domains, including deep neural networks (DNNs) [26, 20], augmented reality and virtual reality (AR/VR) [10], and computational biology [9].

However, on-chip area and power limits impose a constraint on the total number of accelerators that can be simultaneously implemented on a single SoC. Therefore, to optimize full SoC performance against a target workload, we must select an optimal sub-set of kernels to be synthesized into accelerators, while having others be executed in software by the host processor. To do so, the optimization framework must be able to compare the merits of hardware-accelerating two different programs. Quantifying the gains from acceleration is particularly challenging since it requires estimations of both the accelerator performance and the host CPU performance. Furthermore, such estimations should be not only be accurate but also efficient. For example, consider hardware-software co-design, where the co-design compiler performs an iterative search over the accelerator design search space. Measuring the CPU-accelerator performance from a real machine or a full SoC simulation at each iterative step can be exceeding computationally intensive. On the other hand, building an accurate analytical performance model is known to be challenging, especially given the complex nature of modern CPU architectures and constantly-evolving CPU-accelerator interfaces.

In this work, we propose a learning-based performance model to estimate speedups from hardware acceleration. Unlike prior work that is geared towards estimating the performance of CPUs [19, 30, 22] or specific accelerators [14, 18], we devise a framework that will directly learn the HW/SW

*https://github.com/joshuaminwookang/cherrypick.git

Preprint. Work in progress.

relative performance for each program. We also do not limit the scope of our model to specific applications or programming models (e.g. stencil computations or affine loops). As a first step towards this general-purpose performance model, we build a classifier that labels each program with whether it is expected to achieve a relatively large speedup from running on HLS-generated accelerators, compared to other similarly-sized programs.

In summary, this work makes the following contributions:

- We attempt to develop the first learning-based performance model for hardware acceleration speedup, combining two significantly challenging tasks of estimating CPU performance and accelerator performance.
- We implement six classifiers including a random forest classifier, gradient boosting classifier and a multilayer perceptron (MLP) and achieve up to 76.7% test accuracy.
- We generate a dataset based on 12K randomized programs with 56 static program features and hardware/software latency measurements.

2 Related Work

Learning-Based Compiler Optimization. A number of recent works in program analysis and compiler optimization have successfully utilized machine learning models to replace heurstics and analytical models [27, 2, 17, 5, 7]. In particular, there has been a great emphasis on designing autotuners that use learning models to improve various compiler optimization tasks including phase-ordering [11], scheduling [4, 1], hardware prefetching [29] and SIMD/vectorization [6, 8].

Learning Performance Models. Ithemal uses a hierarchical recurrent neural network (RNN) to learn the throughput of modern out-of-order processors. Limiting its scope to basic blocks (BBs), which are short, loop-free sequences of instructions, Ithemal is capable of making estimations that are significantly closer to the ground truth than state-of-the-art analytical models, such as llvm-mca and Intel IACA. Similar learning-based approaches have been attempted by [30] and [22] to estimate CPU performances and for well-known accelerators such as the Tensor Processing Unit (TPU) [14].

Analytical Models for CPU and Accelerator Performance Prediction. OSACA [16] is an open-source static code analyzer based on the LLVM framework [15] that can achieve higher accuracy in runtime prediction compared to llvm-mca and IACA. Aladdin [25] is a pre-RTL simluator that generates dynamic data dependence graphs (DDDG) from high-level programs to represent accelerator designs and make predictions on power and performance. gem5-SALAM [23] improves the accuracy and execution runtime of Aladdin by implementing a dynamic LLVM-based runtime execution engine.

3 Methods

In this work, we consider an accelerator-based SoC system that is capable of executing a program either in the CPU (with software) or through a HLS-generated hardware accelerator. The target task our learning model is to classify a given C program based on its estimated speedup from using accelerators.

3.1 Data Generation

Our dataset was generated using the Csmith [28] randomized test-case generator. Programs generated with Csmith are not only valid C programs without any undefined or unspecified behaviors but are also highly expressive including a variety of common C language features, such as function definitions, control flows and signed/unsigned integer operations. Csmith generated programs often include atypical combinations of such language features and are thus well suited for stress-testing compilers. Therefore, a large dataset of Csmith programs is an appropriate benchmark for our general-purpose program classifier. We produce a dataset of 12K Csmith randomized programs.

Given the source code, we further extracted static program features for each program that include counts of basic blocks (BBs) with certain characteristics and numbers of different types of machine instructions. We use the LLVM framework [15] to extract the features in the same manner Huang

et al. performed their analysis for AutoPhase [11]. The summary of all 56 features is found below in Figure 1. This set of features is comprehensive enough to fairly represent the program and is commonly used by compilers and static schedulers to make decisions about program optimization. We use these 56 features in implementing all of our learning models.

0	Number of BB where total args for phi nodes >5	28	Number of And insts
1	Number of BB where total args for phi nodes is [1,5]	29	Number of BB's with instructions between [15,500]
2	Number of BB's with 1 predecessor	30	Number of BB's with less than 15 instructions
3	Number of BB's with 1 predecessor and 1 successor	31	Number of BitCast insts
4	Number of BB's with 1 predecessor and 2 successors	32	Number of Br insts
5	Number of BB's with 1 successor	33	Number of Call insts
6	Number of BB's with 2 predecessors	34	Number of GetElementPtr insts
7	Number of BB's with 2 predecessors and 1 successor	35	Number of ICmp insts
8	Number of BB's with 2 predecessors and successors	36	Number of LShr insts
9	Number of BB's with 2 successors	37	Number of Load insts
10	Number of BB's with >2 predecessors	38	Number of Mul insts
11	Number of BB's with Phi node # in range (0,3]	39	Number of Or insts
12	Number of BB's with more than 3 Phi nodes	40	Number of PHI insts
13	Number of BB's with no Phi nodes	41	Number of Ret insts
14	Number of Phi-nodes at beginning of BB	42	Number of SExt insts
15	Number of branches	43	Number of Select insts
16	Number of calls that return an int	44	Number of Shl insts
17	Number of critical edges	45	Number of Store insts
18	Number of edges	46	Number of Sub insts
19	Number of occurrences of 32-bit integer constants	47	Number of Trunc insts
20	Number of occurrences of 64-bit integer constants	48	Number of Xor insts
21	Number of occurrences of constant 0	49	Number of ZExt insts
22	Number of occurrences of constant 1	50	Number of basic blocks
23	Number of unconditional branches	51	Number of instructions (of all types)
24	Number of Binary operations with a constant operand	52	Number of memory instructions
25	Number of AShr insts	53	Number of non-external functions
26	Number of Add insts	54	Total arguments to Phi nodes
27	Number of Alloca insts	55	Number of Unary operations

Figure 1: All 56 static program features obtained from LLVM.

Finally, we measure the hardware and software execution times of each Csmith program to complete our dataset generation. Hardware cycle estimation was carried out using LegUp [3] v.4.0 which is an open-source version. We set the target accelerator platform to be Intel's Cyclone-V DE1-SoC board (5CSEMA5F31C6) with Tiger SDRAM. The Cyclone-V FPGA consists of 85K programmable logic elements and is built in the TSMC 28nm low-power process. We set the base clock frequency to be 200MHz, pass the programs through LLVM with `-O3` for preliminary optimization, and then run the LegUp static timing analysis to produce cycle estimations of synthesized accelerators. On the other hand, we measure the CPU(software) execution times of Csmith programs on an Intel I7-1068NG7 processor with 10nm technology and clock frequency 2.3 GHz. Each program was first compiled into binary with `gcc 4.2.1` with the `-O3` flag and measured in Python 3.9 using the `time.time()` function.

3.2 Classification Problem Formulation

In all 12K Csmith random programs, we observe that the hardware accelerator performance is significantly better than software binary performance on CPU. Therefore, the original classification problem we had proposed—to classify whether a program can be accelerated or not—will always be answered "yes" with our dataset. There are several technical nuances as to why this happened, but the biggest reason is that the hardware execution time only includes the cycles spent by the accelerator alone, whereas running program through an accelerator on an actual CPU-FPGA SoC will also incur costs of both control and data communication between the CPU and accelerator. Furthermore, the software execution time measured with Python includes the O/S overheads and time spent context-switching and performing other background tasks on the CPU.

Therefore, we alter the classification problem to consider the relative magnitudes of hardware acceleration. That is, we want to predict whether a given program would achieve a *greater-than-average* performance speedup compared to other programs. To do so, we first need to measure the average speedup that can be expected from our instrumentation setup. Figure 2 shows the total distribution of hardware acceleration speedups on our Csmith dataset. We see that approximately

50% of the random programs achieve greater than 2800× speedup from using HLS accelerators. Hence, we set 2800× as the standard for classifying which subset of programs should be prioritized in synthesizing custom accelerators.

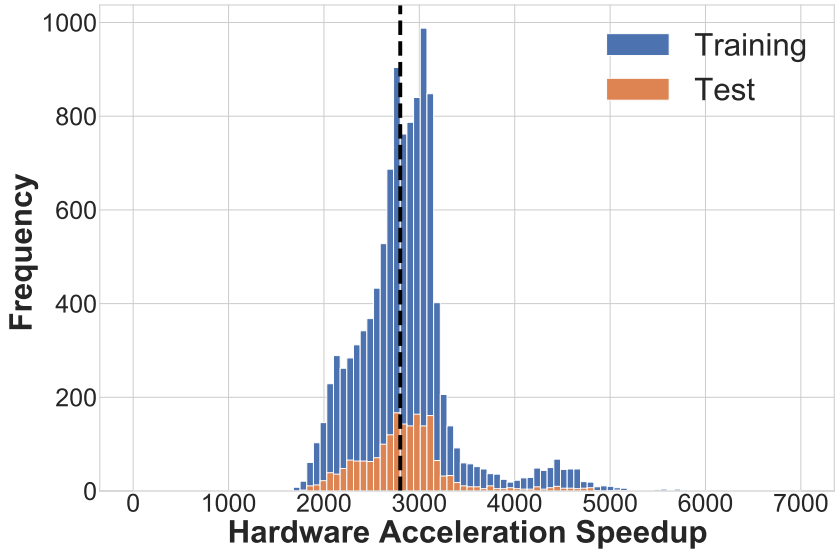

Figure 2: Histogram of speedup distribution for both training and test data. Both datasets are partitions of the original Csmith random program dataset. Dotted vertical line indicates the location of speedup of 2800×. Training and test data was randomly split with a 85%-15% ratio. Notice the similarity of speedup distribution for both training and test.

3.3 Model Implementations

We implement a number of classifiers for our problem to see which learning model performs better or worse. We implement logistic regression, random forests, boosting classifiers (AdaBoost and gradient boosting), and finally, a neural network. All implementations are written in Python3, mainly leveraging Sci-Kit Learn and PyTorch [21] libraries. For all models except for the neural network, we use Sc-Kit Learn's built-in cross-validation score function, so there is no need to nominally set out a separate validation set—therefore we use a 85%-15% split on the original 12K dataset to produce training and test data. For neural networks, we instead perform a 70%-15%-15% split, so that the test dataset is kepy equal among all six learning models.

Here, we discuss hyperparameter tuning results obtained from grid search and our final parameter selections for each model. In each case, the combination of parameters that give the highest average 3-fold cross validation accuracy are chosen as best parameter values.

Logistic Regression. The two parameters at hand are the inverse regularization strength C and the type of norm for penalization. Using the L2 penalty and $C = 0.01$ gives the best result.

Hyperparameters	List of Values	Best Parameter Value
C (Inv. Regularization)	[0.001, 0.01, 0.1]	**0.01**
penalty	[L1, L2, elasticnet]	**L2**

Table 1: Grid search results on logistic regression.

Support Vector Classifier. Similar to logistic regression, a key hyperparameter for support vector classifiers is the inverse regularization strength C. Implicitly, all of our SVM implementations with Sci-Kit Learn use the L2 penalty. Also, we investigate the use of all supported pre-defined kernels—RBF, linear, poly, sigmoid—but poly and sigmoid result in invalid (timeout) results. The shrinkage heuristic is applied to all cases below in Table 2 and there were no limitations set on the number of iterations.

Hyperparameters	List of Values	Best Parameter Value
kernel	[RBF, Linear]	RBF
C (Inv. Regularization)	[0.5, 1, 5]	**1**
γ (kernel parameter)	[0.01, 0.05, 0.1]	**0.01**

Table 2: Grid search results on SVM.

Random Forest Classifier. We search for the best implementation while varying the maximum tree depth, minimum number of samples per leaf, minimum number of samples required for a split and the total number of trees in the forest. In all cases, we use Gini impurity as the split criterion and set the maximum number of features as the square root of the total features. Given 10K training data, the best results are obtained using maximum tree depth of 10 and 450 trees in the forest.

Hyperparameters	List of Values	Best Parameter Value
max_depth	[5, 10]	**10**
min_samples_leaf	[15, 20, 25, 30]	**25**
min_samples_split	[2, 3, 4]	**4**
n_estimators	[300, 350, 400, 450,500]]	**450**

Table 3: Grid search results on random forests.

Boosting Classifiers: AdaBoost and Gradient Boosting. We implement both AdaBoost and Gradient Boosting classifiers and vary their number of estimations, learning rate, and other hyperparmaters. The best case results are summarized in Tables 4 and 5.

Hyperparameters	List of Values	Best Parameter Value
learning_rate	[0.001, 0.01, 0.1]	**0.1**
n_estimators	[400, 500, 6000]	**600**

Table 4: Grid search results on AdaBoost.

Hyperparameters	List of Values	Best Parameter Value
learning_rate	[0.001, 0.01]	**0.01**
max_depth	[5,10]	**5**
criterion	[Friedman_MSE, MAE]	**Friedman_MSE**
subsample	[0.5, 1.0]	**0.5**
n_estimators	[200,300]	**200**

Table 5: Grid search results on gradient boosting classifier.

Multilayer Perceptron (MLP). We implement a MLP neural network in PyTorch by converting our data matrices into tensors. We use SGD with epoch 500 and learning rate 0.001 as our optimizer. We use PyTorch's `BCEWithLogitsLoss` which combines a sigmoid layer and BCE loss togther for back-propagation. Table 6 summarizes the different widths, depths and activation functions used (sigmoid activation was attempted was sub-optimal), from which a 2-layer MLP with width 32 and ReLU activation is selected for best cross-validation accuracy.

Hyperparameters	List of Values	Best Parameter Value
width	[16, 32, 64]	**32**
depth	[2, 3, 4]	**2**
activation function	[ReLU, Tanh]	**ReLU**

Table 6: Grid search results on neural networks.

4 Results

We compare the performance metrics of the best case implementations of all six models. As test data set, we use the remaining 15% of the original Csmith random program dataset we had set aside earlier. 3-fold cross validation accuracy, along with the standard performance measures (accuracy, precision, recall and F1 score) are summarized in Table 7.

	LogReg	SVM	Random Forest	AdaBoost	Grad. Boost	MLP
Cross-Val. Acc.	0.770	0.769	0.773	0.769	0.774	0.774
Test Acc.	0.763	0.759	0.767	0.763	0.767	0.764
Precision	0.737	0.741	0.740	0.741	0.737	0.736
Recall	0.873	0.877	0.880	0.866	0.885	0.880
F1 Score	0.799	0.798	0.804	0.798	0.804	0.802
Confusion Matrix	$\begin{pmatrix} 561 & 325 \\ 133 & 912 \end{pmatrix}$	$\begin{pmatrix} 550 & 336 \\ 129 & 916 \end{pmatrix}$	$\begin{pmatrix} 562 & 324 \\ 125 & 920 \end{pmatrix}$	$\begin{pmatrix} 569 & 317 \\ 140 & 905 \end{pmatrix}$	$\begin{pmatrix} 556 & 330 \\ 120 & 925 \end{pmatrix}$	$\begin{pmatrix} 550 & 331 \\ 126 & 924 \end{pmatrix}$

Table 7: Final result comparison.

Our results reveal several intriguing properties of our classification problem and the Csmith program dataset. First of all, we notice that all six models yield similar test accuracies. A closer look into the confusion matrices also show that the performances from all of our models are roughly equivalent.

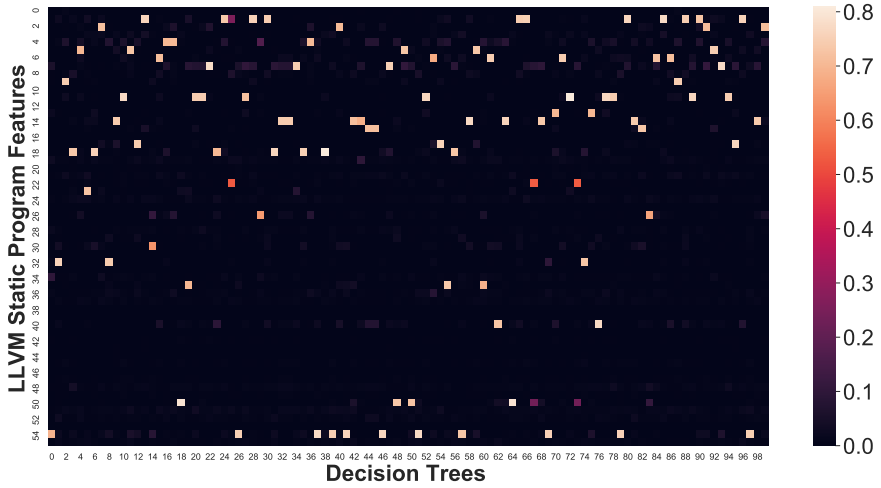

Figure 3: Heatmap of feature importances calculated based on Gini impurity from decision trees included in our final random forest implementation. For readability, we only present 100 trees randomly chosen out of 450 total in the forest.

Considering the current test accuracy (~ 0.76), we suspect there are a few significant latent dimensions that our current set of static features fail to capture. To further investigate the effectiveness of each of our features, we plot a heatmap of feature importances reported from a random subset of decision trees in our final random forest implementation, as displayed in Figure 3. Each feature importance is measured as the normalized total reduction of Gini impurity due to that feature. We observe that many features have consistently low importance while several features relating to the properties of the basic blocks in the program are frequently influential.

More importantly, we observe in Figure 4 that the st features relating to PHI nodes (features 1, 50, 54, and 11-14) are significant in many trees. PHI nodes are LLVM intermediate representation (IR)

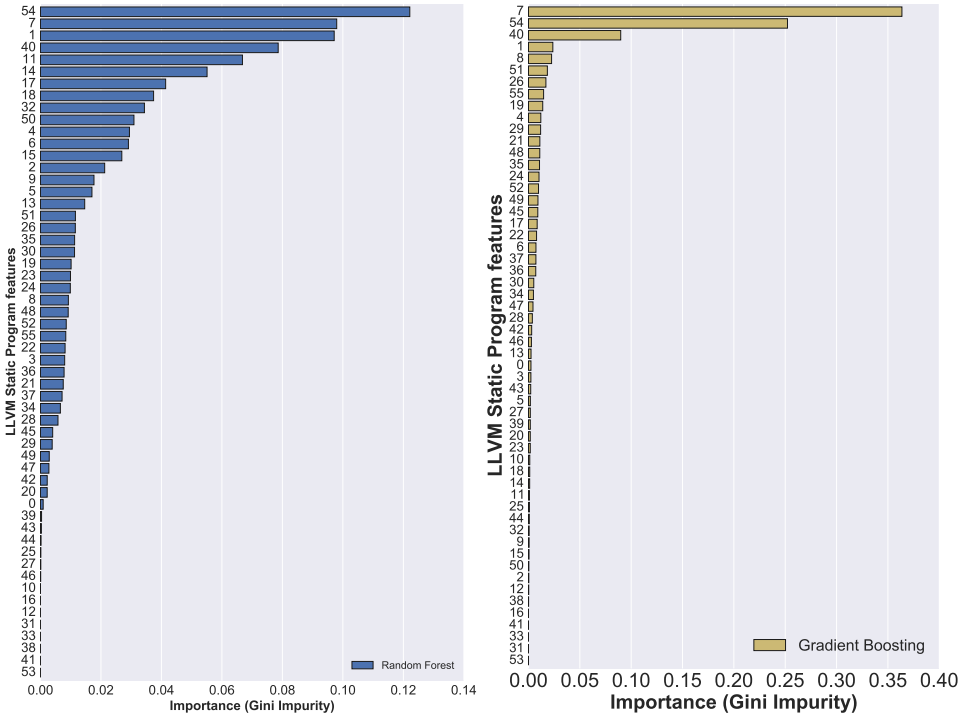

Figure 4: Feature importances reported by random forest and gradient boosting classifiers.

instructions that represent cases where assignments to variables depend on the control flow, such as in an `if` block. More often the program dataflow depends on control logic, it is less likely that the program can be easily mapped into an efficient parallel circuitry. Hence, it is not surprising that program features relating to the number and existence of PHI nodes highly influence the magnitude of hardware acceleration of each program.

On the other hand, we also suspect that our test accuracy may be improved if we used a deeper neural network, perhaps with hierarchical architectures. It is possible that a more complex function approximation would better model the relationship between speedup and our static features. Based on the examples of Ithemal [19] and the TPU performance model [14], we predict that a graph-based neural network or a hierarchical RNN implementation of our classifier would return improved performances. We plan to further investigate this idea, along with others discussed in the next section, in the near future.

5 Conclusion and Future Work

In this work, we have presented a first step towards learning the HW/SW relative performance model for general-purpose programs. We first generated an expansive dataset consisting of randomized programs with their hardware and software execution time measurements and LLVM static program features. From our evaluation of implemented classifiers, we have shown that we can achieved up to 76.7% test accuracy and 88.5% sensitivity (recall) with relatively simple and compute-efficient learning models. A subset of the our static features are shown to be ineffective in predicting the hardware speedup and discussed reasons to investigate deeper neural network models to further improve classifier performance.

There are several immediate plans to extend and improve this work. One is to re-visit the data generation pipeline to more accurately measure the hardware execution latency using full SoC simulation platforms, such as FireSim [13] and Centrifuge [12]. At the same time, we expect to improve the CPU timing infrastructure as well by eliminating O/S overhead from our measurements. We plan to directly insert timing instructions into our dataset programs and thus into their compiled binaries, instead of measuring execution time from an external Python script. We also plan to explore

the viability of building a regression model instead of a classifier or to classify with k-labels where each label indicates the varying degrees of speedup/slowdown in hardware. Finally, we plan to expand our dataset by generating more Csmith randomized programs and also include other benchmark suites to ensure that we have sufficient amounts of data to later train deeper neural networks.

A more long-time research direction is to investigate the robustness of the learned acceleration performance models and their susceptibility to outliers and changes to the hardware architecture assumptions. For example, it will intriguing to observe how modifications to configurations of either the host processor or the accelerator substrate would affect the classifier performance. Constructing a model with high cross-platform performance will be a challenging yet extremely rewarding research goal.

References

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4):1–12, 2019.

[2] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5), September 2018.

[3] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, page 33–36, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[5] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[6] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE, 2008.

[7] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, 2019.

[8] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. Neurovectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 242–255, 2020.

[9] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H. Oh, Krste Asanovic, Jae W. Lee, and Lisa Wu Wills. Genesis: A hardware acceleration framework for genomic data analysis. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 254–267. IEEE Press, 2020.

[10] Dukki Hong, Tae-Hyoung Lee, Yejong Joo, and Woo-Chan Park. Real-time sound propagation hardware accelerator for immersive virtual reality 3d audio. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek. Autophase: Compiler phase-ordering for hls with deep reinforcement learning. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 308–308, 2019.

[12] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. Centrifuge: Evaluating full-system hls-generated heterogenous-accelerator socs using fpga-acceleration. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[13] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: FPGA-Accelerated Cycle-Exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[14] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, and Mike Burrows. A Learned Performance Model for the Tensor Processing Unit. *arXiv e-prints*, page arXiv:2008.01040, August 2020.

[15] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[16] Jan Laukemann, Julian Hammer, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131. IEEE, 2018.

[17] Yi Li, Shaohua Wang, and Tien N Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[18] Hosein Mohammadi Makrani, Hossein Sayadi, Tinoosh Mohsenin, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. Xppe: cross-platform performance estimation of hardware accelerators using machine learning. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 727–732, 2019.

[19] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, Jun 2019. PMLR.

[20] Surya Narayanan, Karl Taht, Rajeev Balasubramonian, Edouard Giacomin, and Pierre-Emmanuel Gaillardon. Spinalflow: An architecture and dataflow tailored for spiking neural networks. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 349–362. IEEE Press, 2020.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[22] A. Renda, Y. Chen, C. Mendis, and M. Carbin. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 442–455, 2020.

[23] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi. gem5-salam: A system architecture for llvm-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482, 2020.

[24] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Yakun Sophia Shao, Sam Likun Xi, Vijayalakshmi Srinivasan, Gu-Yeon Wei, and David Brooks. Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[26] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[27] Z. Wang and M. O'Boyle. Machine learning in compiler optimization. *Proceedings of the IEEE*, 106(11):1879–1901, 2018.

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[29] Yuan Zeng and Xiaochen Guo. Long short term memory based hardware prefetcher: a case study. In *Proceedings of the International Symposium on Memory Systems*, pages 305–311, 2017.

[30] M. Zhou, J. Chen, H. Hu, J. Yu, Z. Li, and H. Hu. Deeptle: Learning code-level features to predict code performance before it runs. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 252–259, 2019.