Neuroevolutionary Compiler Control for Code Optimization

Kade Heckel
kade.heckel@gmail.com
University of Sussex
Brighton, East Sussex, United Kingdom

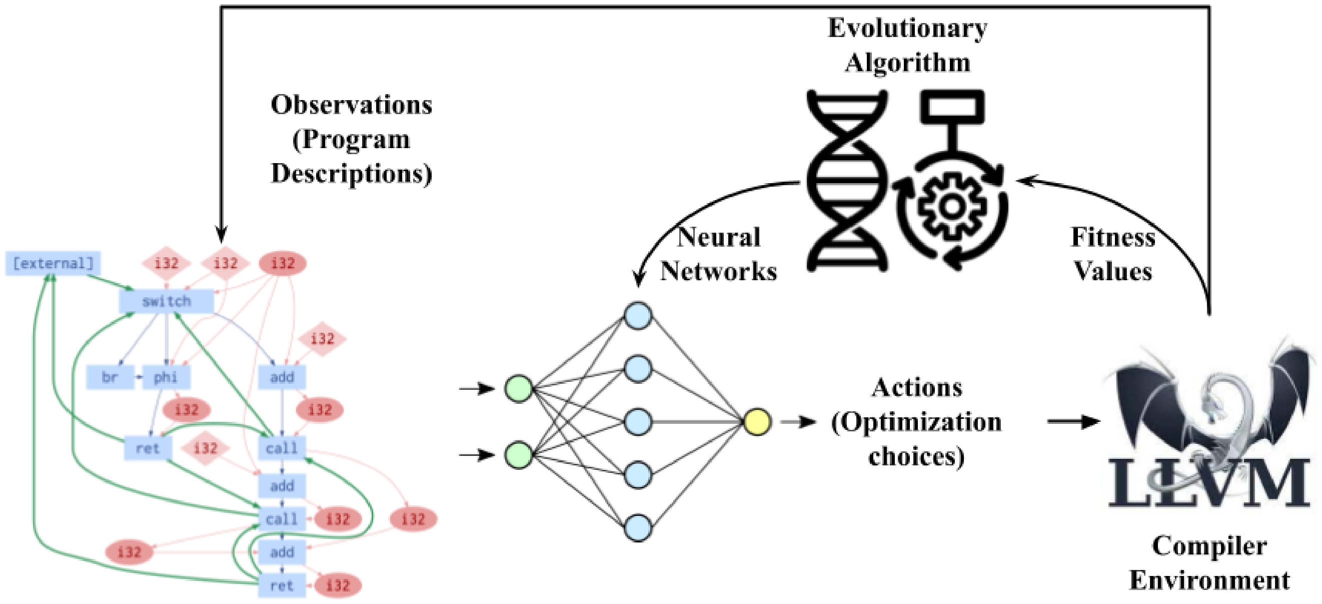

Figure 1. Neuroevolutionary Compiler Control is an adaptive system where feedback drives selection of higher-performing neural networks.[1–4]

Abstract

The optimization performed by compilers when generating executable programs is critical for software performance yet tuning this process to maximize efficiency is difficult due to the large number of possible modifications and the almost limitless number of potential input programs. To promote the application of artificial intelligence and machine learning to this challenge, Facebook Research released Compiler Gym[4], a reinforcement learning environment to allow the training of agents to perform compiler optimization control on real C/C++ programs. Whereas previously published approaches use techniques such as Proximal Policy Optimization or Deep Q Networks, this work utilizes neuroevolution and achieves competitive performance on the cBench-v1 program set[5] while demonstrating the highly adaptive properties of the neuroevolution approach.

Keywords: Neuroevolution, Compilers

1 Introduction

The code written by a programmer is not directly understandable by a computer; it must either be compiled into binary machine instructions to be executed on the processor or interpreted by another compiled program called an interpreter. The compilation process ostensibly has five stages: lexical analysis, parsing/syntax checking, abstract syntax tree generation, code generation, and code optimization. Lexical Analysis begins by tokenizing the source code into sequences such as string keywords, variables, numbers, strings, and other symbols such as brackets and parentheses. These tokens are then parsed to ensure the syntax is correct, such as ensuring operators have the proper number of operands and that statements are complete. These parsed statements are turned into an abstract syntax tree which represents the computation structure of the program. From this point, code generation occurs for the specific target architecture; in the case of the LLVM compiler environment the abstract syntax tree is turned into an intermediate representation (IR) which is similar to assembly code with some exceptions such as single-static assignment where variables are not reused. This

IR code is then mapped to registers and architecture specific instructions after several optimization passes are performed. These optimization passes modify sections of the IR to reduce code size and/or increase efficiency through techniques such as loop unrolling or function in-lining. Converted to machine code, the program can then be executed on the target architecture.[6]

The process of applying code optimizations is a complex process currently guided by heuristics. To encourage exploration of the task through the use of artificial intelligence, Facebook AI Research released Compiler Gym, a reinforcement learning environment for training agents to control the compiler code optimization process. CompilerGym converts the compiler optimization process into a sequential decision task of selecting and applying optimizations to a program with the reward signal being relative to the code size reduction achieved by LLVM's default optimization settings. The top approaches on the Compiler Gym leaderboard include PPO-directed search and a large-scale brute-force random search, with the investigation of a graph neural network for processing program observations being another noteworthy mention. [4] In this work, neuroevolution was used for the first time to evolve a population of 40 multilayer-perceptrons to direct the compiler optimization process.

Evolutionary computation has not been investigated with respect to compiler optimization control. These nature-inspired methods rely on mutating and adapting a population of potential solutions to a problem, performing stochastic optimization without need for the underlying task to be differentiable. Neuroevolution, the application of evolutionary approaches to neural networks, has gained popularity as a viable alternative to gradient based methods in deep learning due to the fact that rewards may be sparse and the gradient uninformative.[7] Given that only a few actions at a time may alter the program structure beneficially, the largely parallel nature of neuroevolution poses a benefit for rapidly identifying beneficial actions compared to relying on a single network to learn from a large action space. The use of neuroevolution in reinforcement learning constitutes an adaptive system in which the population of neural networks and the corresponding action distribution changes over time based on observations and feedback from the environment in terms of a reward signal. This aligns with the definition of an adaptive system per Mayr: "a property of an organism, whether a structure, a physiological trait, a behavior, or anything else that the organism possesses, that is favored by selection over alternative traits."[8] In the case of this work, the deterministic behaviors exhibited by the neural networks in the population undergo selection and adapt toward better program optimizations, with the LM-MA-ES algorithm and CompilerGym environment driving the adaptive system illustrated in Figure 1. Evolutionary approaches differ from gradient-based deep reinforcement learning techniques such as Proximal Policy Optimization in that a population of solutions is used rather than a pair of networks which estimate the values of the agent's predicted actions[9].

Three datasets were used in experimentation: CHStone, cBench, and a sample of 10 BLAS programs. CHStone is a set of 12 large C programs from various application domains presented as a benchmark for high-level synthesis research for hardware. Application domains include arithmetic, media processing, cryptography, and more, providing a varied set of program types[10]. The cBench dataset which contains a larger set of programs similar in composition to CHStone was released as a benchmark suite for collective and distributed optimization of computer programs[5]. Finally, 10 programs from BLAS are used to test the generalization and adaptability of the evolved neural networks, as the BLAS kernels are extremely dense in arithmetic operations and are sparse on memory access operations unlike the CHStone and cBench sets[11][4].

2 Results

This work finds that neuroevolution is competitive with deep reinforcement learning techniques such as Proximal Policy Optimization while taking significantly less time to train. Further analysis showcases the strong generalization and adaptability of the models evolved using the LM-MA-ES algorithm and investigates the relationships between programs in the dataset and the corresponding actions performed by the network population. Critically, the approach is highly sample efficient as the best evolved network surpasses the default LLVM compiler optimization settings in under 100 program Evaluations otherwise known as rollouts.

Figure 2 illustrates the best fitness per epoch over the three datasets for three different values of k, which controls how many actions the neural network selects at a time. It can be seen that larger values of k correspond both with higher achieved fitness values as well as adaptability to new datasets. The maximum fitness achieved on the cBench dataset was 1.045799, which is competitive with the 1.047 achieved by using a Graph Attention Network and Decentralized Distributed Proximal Policy Optimization and is not far off the 1.07 state of the art result achieved using Proximal Policy Optimization and Guided Search.[4] Figure 3 shows the wall time for executing 15 passes through each dataset, showing that the cBench dataset takes substantially longer to evaluate. This is a result of cBench having 23 programs versus 12 or 10 for the CHStone or BLAS sets as well as the fact that the ghostscript program unique to cBench is exceptionally intensive to compile; another entry on the CompilerGym leaderboard set it out as a validation point and trained on the others to accelerate the learning process. Notably, the neuroevolution approach was able to learn from the CHStone dataset and generalize with minimal performance loss to the cBench dataset in aprrroximately 5 minutes, drastically improving on the 7 hours needed to train a PPO model to assist guided search in the state-of-the-art approach. [4]

Figures 4 through 16 present a similarity analysis of the program instruction vectors and the evolved neural network population action distributions to illustrate the adaptation of the networks to families of programs and also be able to apply distinct transformations to sub-categories.

Figures 4, 5, and 6 show the intradataset instruction vector cosine distance for the CHStone, cBench, and BLAS datasets respectively. The intradataset similarity values are interesting as they visualize the diversity of the observation landscape the evolved networks will be presented with. The interdataset similarity between the CHStone and cBench datasets and the dissimilarity with the BLAS programs is depicted in figures 7 and 8, where the cosine similarity is 30% lower when comparing cBench to BLAS vs cBench and CHStone. These instruction count cosine similarity plots of support the distinction between the first two sets and the BLAS programs in terms of general purpose vs arithmetic programs having differing characteristics. [4]

Figures 9, 10, and 11 show the distribution of actions selected by the neural network population when $k = 10$. It can be seen that distinct spectra emerge for each dataset, where the CHStone/cBench spectra have some overlap but differ significantly from the BLAS spectra. The top 5 actions selected by each network are listed in table 1.

Figures 12, 13, and 14 depict the intradataset cosine similarities for the distributions of actions selected by the evolved population. While the range in similarty values is rather small, it is noteworthy that the correlations correspond with the instruction count similarities depicted in figures 4, 5, and 6, showing that the evolved networks do distinguish between programs when making optimization decisions.

Finally, figures 15 and 16 showcase the interdataset cosine similarities of the action distributions for the three datasets and follows the trend from figures 7 and 8 where the cBench to BLAS comparison has much lower similarity scores compared to CHStone versus cBench.

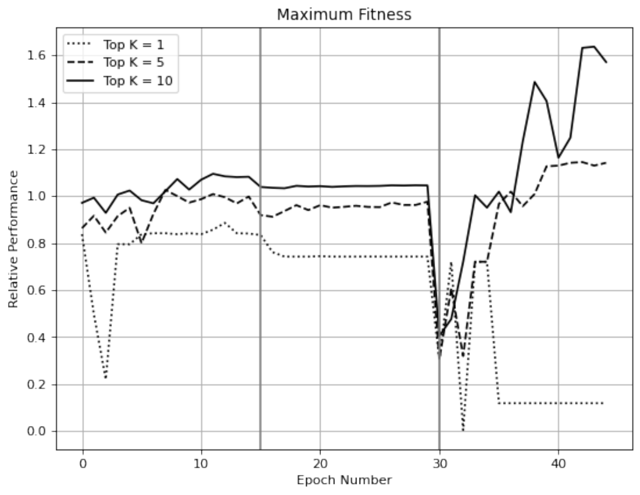

Figure 2. Best fitness attained per epoch. Vertical lines at epochs 15 and 30 denote transition from the CHStone-v0 to cBench-v1 and cBench-v1 to BLAS-v0 respectively.

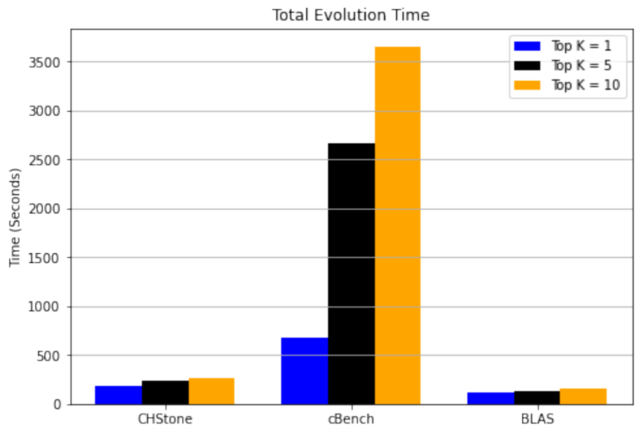

Figure 3. Wall execution time grouped by dataset and colored by the number of actions recommended per query of the evolved neural network.

3 Discussion and Future Work

3.1 Conclusions

The novel application of neuroevolution to compiler optimization control found the method to be competitive with gradient based deep reinforcement learning techniques while being computationally and sample efficient. The results indicate that a neuroevolutionary approach is able to rapidly adapt to the CompilerGym task and achieve high performance with behavior adapted to specific program types, and when the program distribution shifts to a previously unseen set as was the case with the cBench to BLAS transition the population is able to recover and return to high performance solutions.

It was found that converting the problem from a pure sequential decision problem back into a batched decision problem similar to how compilers bundle together methods into optimization passes was drastically beneficial to the adaptation process for the evolved neural networks. Given an action space of size 124 and a sparse reward signal resulting from few actions being effective for a given program composition, the original formulation of picking a single action after another makes it difficult to evolve networks which select multiple valuable actions. By taking the actions corresponding to the top-k highest network activations the task can be seen as the network ranking the top actions based on the program's observed instruction counts; since more actions are tried by each agent it is easier to identify

	CHStone	cBench	BLAS
1	Scalarize vector operations	Scalarize vector operations	Merge Functions
2	Dead Global Elimination	Dead Global Elimination	Global Value Numbering
3	Assign names to anon. instructions	Assign names to anon. instructions	Eliminate Available Externally Globals
4	Lower SwitchInst's to branches	Simplify the CFG	Induction Variable Simplification
5	Promote Memory to Register	Remove unused exception handling info	Lower 'expect' Intrinsics

Table 1. Top 5 action selections per dataset

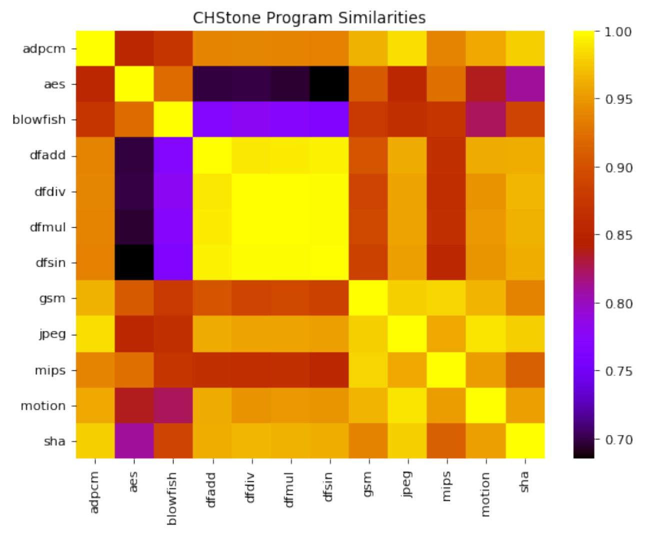

Figure 4. Cosine Similarity between instruction counts in the CHStone dataset. Note that the AES and Blowfish encryption algorithms differ significantly to the dfadd and dfmul programs.

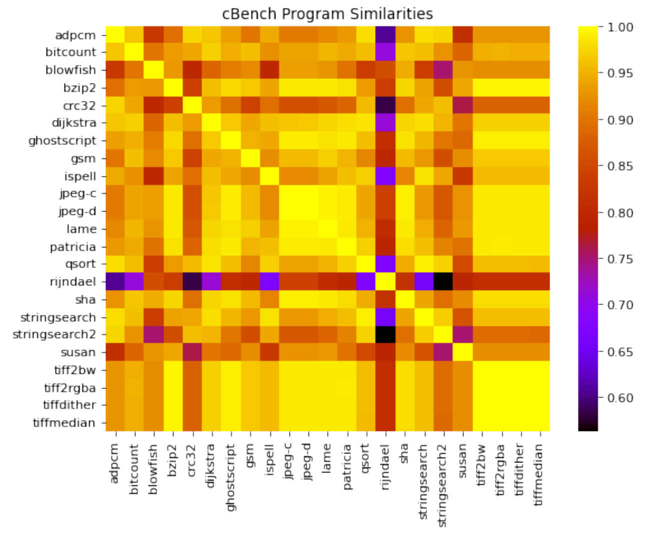

Figure 5. Cosine Similarity between instruction counts in the cBench dataset. Note Rijndael, a more flexible version of AES, differs from the majority of other programs.

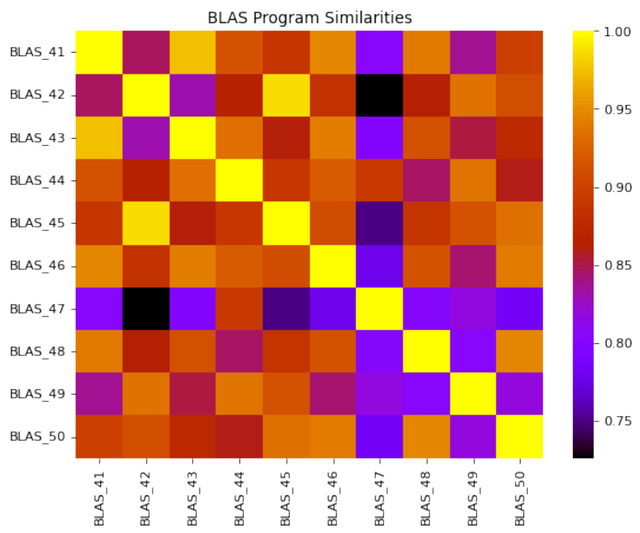

Figure 6. Cosine Similarity between instruction counts in the BLAS dataset. Note that program 47 is an outlier within the set in terms of instruction distribution.

ones which yield reward and improves adaptability of the networks. It can be seen in figure 2 that taking only the highest ranked action results in deleterious effects when the program set is changed whereas taking multiple actions allows for the evolved network populations to adapt to the BLAS dataset.

Based on the analysis of the action distributions and comparing them to the program similarities, it can be seen that the evolved networks adapt their selected actions to program structures. For example, the high similarities in the instruction distributions for the tiff series programs in the cBench dataset correspond with highly similar patterns in the action distributions by the neural network population. The temporal patterns of the adaptation are exemplified in figures 9, 10, and 11 where the action distributions concentrate on small sets of actions which are unique to the particular dataset.

As evidenced by the temporal data in figure 3 and connecting it with the fitness data of figure 2, it can be seen that neural controllers for compiler optimization can be pretrained or pre-evolved on smaller programs with similar instruction counts and then used on larger, more intensive programs to

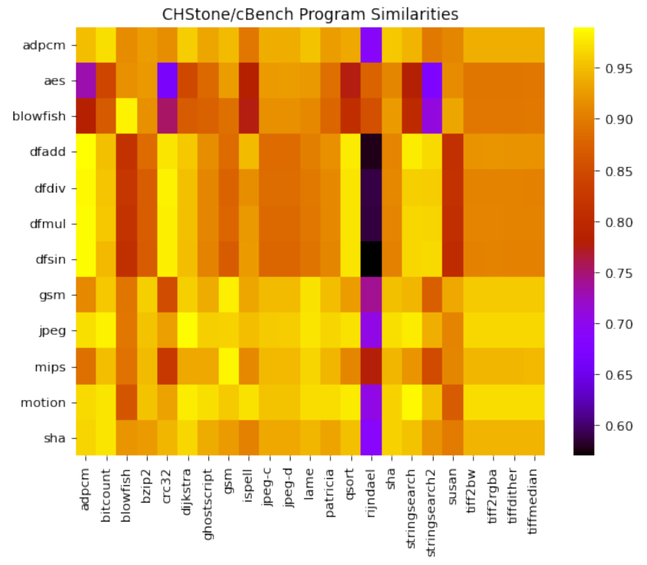

Figure 7. Similarities between the CHStone and cBench datasets; note the high similarity between the two sets with the exception of Rijndael, which is only similar to the AES and Blowfish programs.

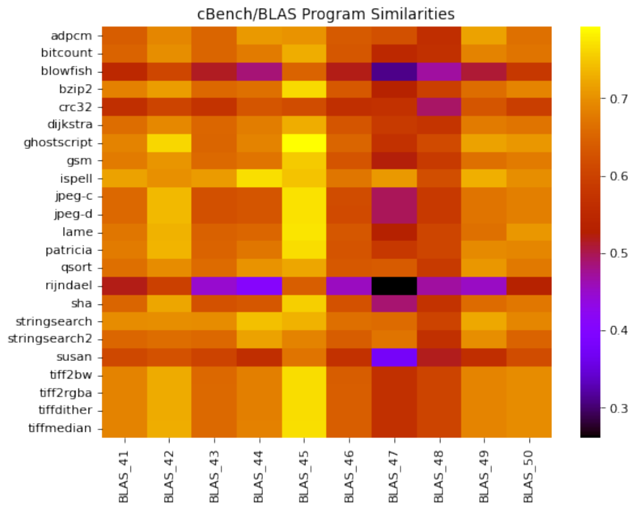

Figure 8. Cosine similarities between the cBench and BLAS datasets. Note the significant drop-off in similarity compared to CHStone-cBench.

compile in a pretraining fashion. This is especially important when noting the $k = 10$ case where the cBench dataset takes 5 times as long to complete 15 epochs on when accounting for the difference in the number of programs in the set.

Finally, the neuroevolution of a graph neural network to process the program structure was investigated unsuccessfully, finding that the computational demand severely

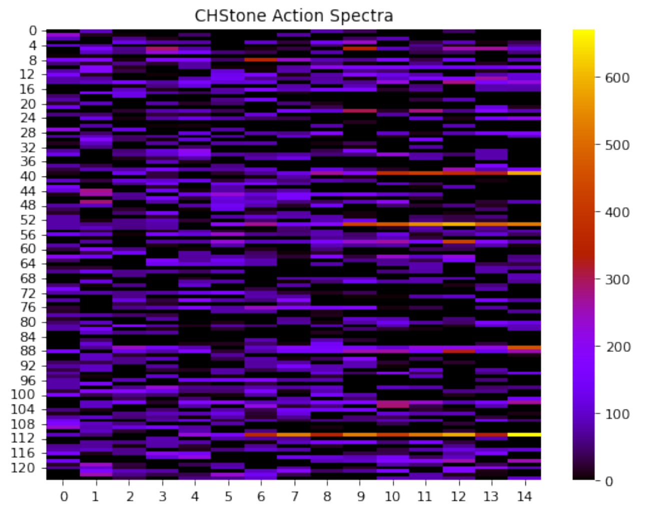

Figure 9. Action distribution spectra on the CHStone dataset for $k = 10$.

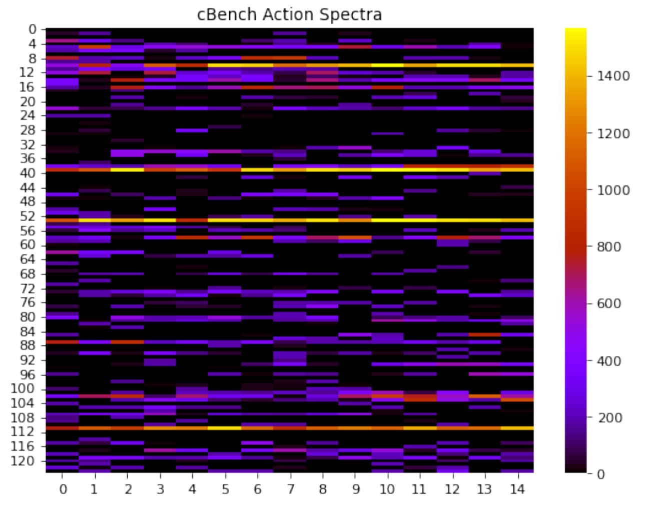

Figure 10. Action distribution spectra on the cBench dataset for $k = 10$.

constrained the possible population size to a few individuals barring performing the evolutionary process on a server with several dozen cores.

3.2 Future Work

Future work could perform a more comprehensive architecture search on the model to be evolved and even seek to apply recurrent neural networks or transformer architectures to the task to boost performance. Recurrent neural networks could resolve the issue of rapidly repeating actions since they can keep an internal representation of previous states; this could enable learning of useful sequences such as optimizing loops and then removing dead code afterwards.

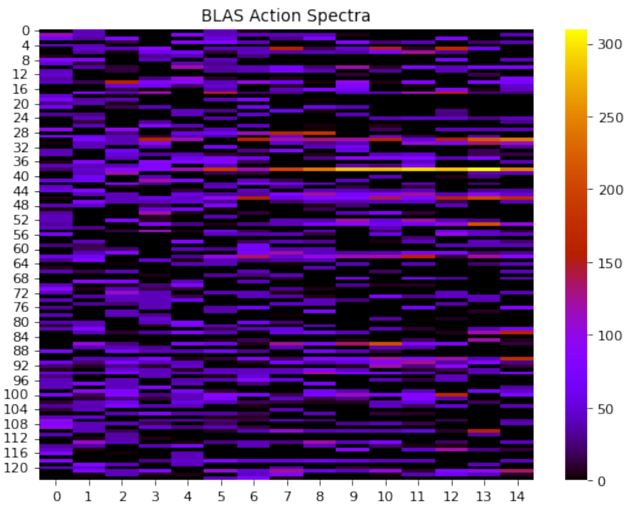

Figure 11. Action distribution spectra on the evaluated subset of the BLAS dataset for $k = 10$.

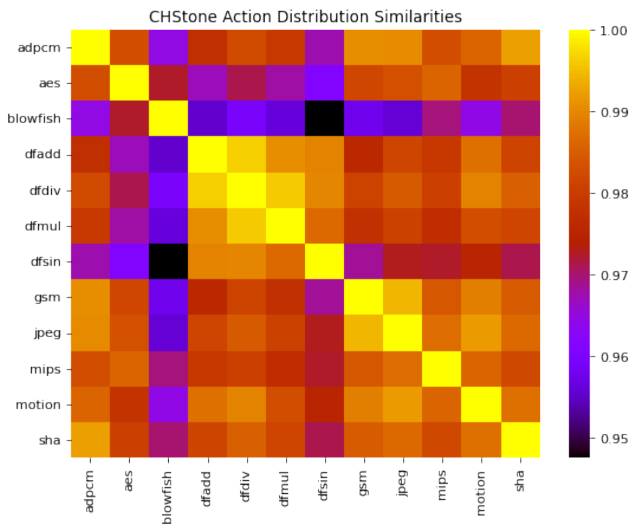

Figure 12. Intradataset action distribution similarities for CHStone. The dissimilarity of Blowfish to all other programs except AES is observed again.

A Tab Transformer could use both the normalized instruction count vectors and the instruction embedding vector observation space provided by CompilerGym[4, 12]. This approach could use ANGHABENCH as it contains over 1 million C programs scraped from public code repositories, providing a vast set of programs to train on[13]. By applying a TabTransformer in a Decision Transformer architecture, a model could be learned that takes the total instruction counts and their embedding vectors and learns state-action-regard trajectories to learn high performance behavior in an offline context[14].

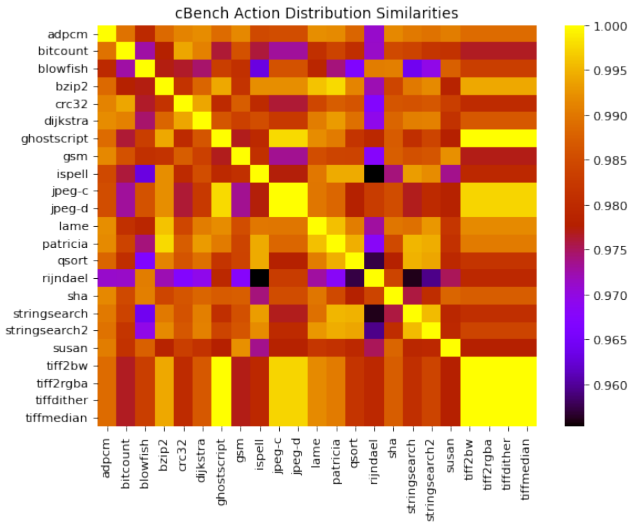

Figure 13. Intradataset action distribution similarities for cBench. Note the general correspondence with the cBench instruction count similarities and the same actions being applied for the series of tiff programs.

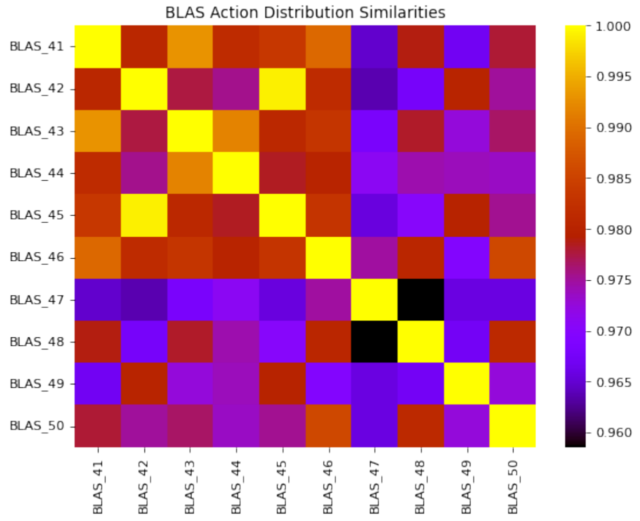

Figure 14. Intradataset action distribution similarities for BLAS. Again note the relation to the instruction vector similarities.

Finally, learning a surrogate model for compilation would also be extremely useful as the training process is CPU bound due to the cost of compiling programs. A surrogate model could help improve the sample efficiency by weeding out bad networks or action choices in advance and enable for faster training speeds.

References

[1] Edivaldo Brito. Arquivos llvm 16.

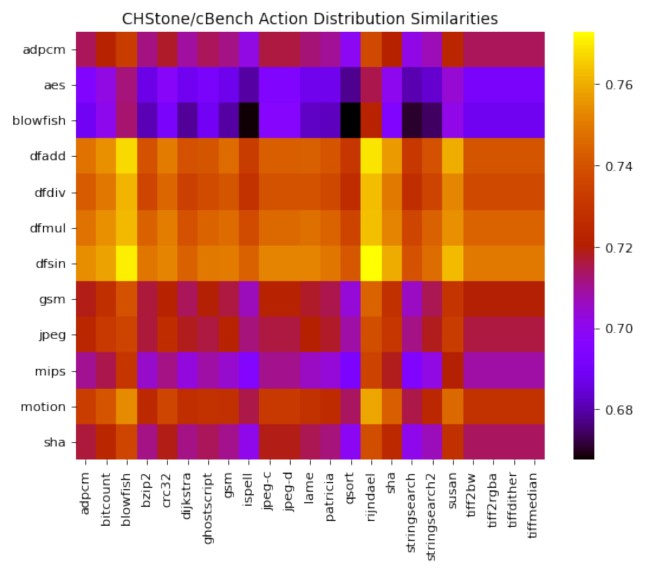

Figure 15. CHStone/cBench action similarities. Interestingly the actions for the arithmetic programs in CHStone have a higher similarity with the Rijndael program in cBench, which is the opposite of what the instruction count similarities would predict.

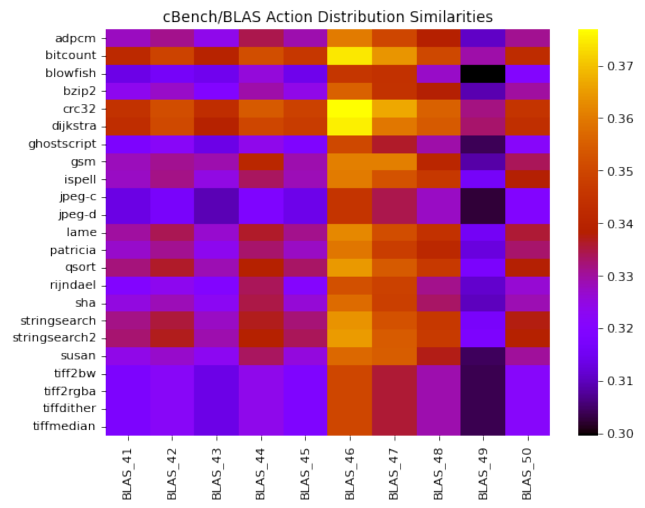

Figure 16. cBench/BLAS action similarities. Again, counter-intuitively, the BLAS programs that were less similar in instruction composition with the cBench programs had the more similar action distributions.

[2] Symbolon. Genetic algorithm icon - free png amp; svg 3263187 - noun project, Feb 2020.

[3] File:neural network.svg.

[4] 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. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *CGO*, 2022.

[5] Grigori Fursin. Collective tuning initiative: Automating and accelerating development and optimization of computing systems. 07 2014.

[6] Stages of compilation.

[7] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR*, abs/1712.06567, 2017.

[8] E. Mayr. *What Evolution Is.* Science Masters Series. Basic Books, 2001.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[10] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *2008 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195, 2008.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, sep 1979.

[12] Xin Huang, Ashish Khetan, Milan Cvitkovic, and Zohar S. Karnin. Tabtransformer: Tabular data modeling using contextual embeddings. *CoRR*, abs/2012.06678, 2020.

[13] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quinão Pereira. 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)*, pages 378–390, 2021.

[14] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *CoRR*, abs/2106.01345, 2021.

[15] Ilya Loshchilov, Tobias Glasmachers, and Hans-Georg Beyer. Limited-memory matrix adaptation for large scale black-box optimization. *CoRR*, abs/1705.06693, 2017.

[16] Robert Tjarko Lange. evosax: Jax-based evolution strategies. *arXiv preprint arXiv:2212.04180*, 2022.

[17] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[18] Lilian Weng. Evolution strategies. *lilianweng.github.io*, 2019.

[19] Nikolaus Hansen. The cma evolution strategy: A tutorial, 2016.

[20] Hans-Georg Beyer and Bernhard Sendhoff. Simplify your covariance matrix adaptation evolution strategy. *IEEE Transactions on Evolutionary Computation*, 21(5):746–759, 2017.

[21] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

A Methodology

The Low Memory Matrix Adaptation Evolution Strategy [15] was employed for evolving neural network controllers utilizing the implementation provided in the EvoSAX Python library[16] which is built on top of the JAX library[17]. LM-MA-ES is an optimized version of Covariance Matrix Adaptation - Evolution Strategy which optimizes a probabilistic model through iteratively generating, evaluating, and discarding sample solutions. As an Estimation of Distribution Algorithm, it differs from classic genetic algorithms by modeling the implicit probability distribution of solutions rather than operating locally on samples of an explicit distribution. That is, rather than sampling a random distribution and then using recombination, mutation, and selection operators to improve solutions, EDAs evolve a probabilistic model through the iterative generation of samples and updating the probabilistic model to improve the quality of future generated samples. Specifically, CMA-ES which builds on CMA generates parameters in accordance with the following equation:

$$\theta = (\mu, \sigma, C), p_\theta(x) \sim \mathcal{N}(\mu, \sigma^2, C) \sim \mu + \sigma\mathcal{N}(0, C))$$

These parameters are sampled from a normal distribution with mean μ and a covariance matrix C which models the interdependence between parameters; the step size σ is used to scale the covariance matrix and functions as a learning rate to adjust the rate at which the population of parameters changes. The step size adapts itself based on the evolution path, which the algorithm constructs by tracking evolutionary steps over time. If the mean barely shifts over several steps it is an indicator that the step size is too large and is bouncing around an optima, and σ is decreased. If the steps in the evolution path are correlated and point in the same direction that indicates the step size is too small and that σ needs to be increased. Traditional CMA updates C by calculating the covariance of a fraction of elite samples and using that to generate the next generation of samples while CMA-ES utilizes more complex updating strategies including a to function with smaller population sizes. To reduce the necessary population size to obtain good estimates of the covariance matrix, a history is used to retain information from previous generations to improve sample efficiency[18, 19].

Further work on optimizing CMA-ES found that the algorithm could be approximated with little performance loss while removing the covariance matrix update and removing the need to perform matrix square root operations on the covariance matrix; this is achieved through the use of a transformation matrix $\mathcal{M}$ that corresponds to the square root of the covariance matrix which describes the population[20]. The Limited-Memory Matrix Adaptation Evolution Strategy algorithm used in this work reduces the time and space complexity of the MA-ES algorithm to $O(nlog(n))$ by requiring the transformation matrix be diagonal and low-rank, allowing M to be scaled to the thousands of parameters[15]. Psuedocode illustrating the differences between CMA-ES, MA-ES, and LM-MA-ES is presented in figure 17[15].

A multilayer perceptron was used as the model for controlling the compiler's optimization decisions. Composed of 20,860 parameters split across 69 input neurons, 3 hidden layers of 64 neurons with Rectified Linear Unit activations, and an output layer of 124 neurons, the networks were evolved to select and order a top-k number of actions for the compiler to carry out using the CompilerGym's multistep functionality. Patience was implemented in the environment wrapper such

```
Algorithm 1 CMA-ES , MA-ES and LM-MA-ES
1: given $n \in \mathbb{N}_+, \lambda = 4 + \lfloor 3 \ln n \rfloor, \mu = \lfloor \lambda/2 \rfloor, w_i = \frac{\ln(\mu+\frac{1}{2}) - \ln i}{\sum_{j=1}^{\mu}(\ln(\mu+\frac{1}{2}) - \ln j)}$ for $i = 1, \ldots, \mu$, $\mu_w = \frac{1}{\sum_{i=1}^{\mu} w_i^2}$, $c_\sigma = \frac{\mu_w+2}{n+\mu_w+5}, c_c = \frac{4}{n+4}, c_1 = \frac{2}{(n+1.3)^2+\mu_w}, c_\mu = \min\left(1 - c_1, \frac{2(\mu_w - 2 + 1/\mu_w)}{(n+2)^2+\mu_w}\right)$,
   $m = 4 + \lfloor 3 \ln n \rfloor, c_\sigma = \frac{2\lambda}{n}, c_{d,i} = \frac{1}{1.5^{i-1} n}, c_{c,i} = \frac{\lambda}{4^{i-1} n}$ for $i = 1, \ldots, m$
2: initialize $t \leftarrow 0, \boldsymbol{y}^{(t=0)} \in \mathbb{R}^n, \sigma^{(t=0)} > 0, \boldsymbol{p}_\sigma^{(t=0)} = \boldsymbol{0}$, $\boldsymbol{p}_c^{(t=0)} = \boldsymbol{0}, \mathbf{C}^{(t=0)} = \mathbf{I}$ , $\mathbf{M}^{(t=0)} = \mathbf{I}$ ,
   $\boldsymbol{m}_i^{(t=0)} \in \mathbb{R}^n, \boldsymbol{m}_i^{(t=0)} = \boldsymbol{0}$ for $i = 1, \ldots, m$
3: repeat
4:   for $i \leftarrow 1, \ldots, \lambda$ do
5:     $\boldsymbol{z}_i^{(t)} \leftarrow \mathcal{N}(\boldsymbol{0}, \mathbf{I})$
6:     $\boldsymbol{d}_i^{(t)} \leftarrow \boldsymbol{z}_i^{(t)}$
7:     if $t \mod \frac{n}{\lambda} = 0$ then $\mathbf{M}^{(t)} \leftarrow \sqrt{\mathbf{C}^{(t)}}$ else $\mathbf{M}^{(t)} \leftarrow \mathbf{M}^{(t-1)}$        ▷ CMA-ES
8:     $\boldsymbol{d}_i^{(t)} \leftarrow \mathbf{M}^{(t)} \boldsymbol{d}_i^{(t)}$        ▷ CMA-ES and MA-ES
9:     for $j \leftarrow 1, \ldots, \min(t, m)$ do        ▷ LM-MA-ES
10:      $\boldsymbol{d}_i^{(t)} \leftarrow (1 - c_{d,j})\boldsymbol{d}_i^{(t)} + c_{d,j}\boldsymbol{m}_j^{(t)} \left((\boldsymbol{m}_j^{(t)})^T \boldsymbol{d}_i^{(t)}\right)$        ▷ LM-MA-ES
11:    $\boldsymbol{f}_i^{(t)} \leftarrow f(\boldsymbol{y}^{(t)} + \sigma^{(t)}\boldsymbol{d}_i^{(t)})$
12:  $\boldsymbol{y}^{(t+1)} \leftarrow \boldsymbol{y}^{(t)} + \sigma^{(t)} \sum_{i=1}^{\mu} w_i \boldsymbol{d}_{i:\lambda}^{(t)}$        ▷ the symbol $i : \lambda$ denotes $i$-th best sample on $f$
13:  $\boldsymbol{p}_\sigma^{(t+1)} \leftarrow (1 - c_\sigma)\boldsymbol{p}_\sigma^{(t)} + \sqrt{\mu_w c_\sigma(2 - c_\sigma)} \sum_{i=1}^{\mu} w_i \boldsymbol{z}_{i:\lambda}^{(t)}$
14:  $\boldsymbol{p}_c^{(t+1)} \leftarrow (1 - c_c)\boldsymbol{p}_c^{(t)} + \sqrt{\mu_w c_c(2 - c_c)} \sum_{i=1}^{\mu} w_i \boldsymbol{d}_{i:\lambda}^{(t)}$        ▷ CMA-ES
15:  $\mathbf{C}^{(t+1)} \leftarrow (1 - c_1 - c_\mu)\mathbf{C}^{(t)} + c_1 \boldsymbol{p}_c (\boldsymbol{p}_c^{(t)})^T + c_\mu \sum_{i=1}^{\mu} w_i \boldsymbol{d}_{i:\lambda}^{(t)} (\boldsymbol{d}_{i:\lambda}^{(t)})^T$        ▷ CMA-ES
16:  $\mathbf{M}^{(t+1)} \leftarrow \mathbf{M}^{(t)} \left[\mathbf{I} + \frac{c_1}{2}\left(\boldsymbol{p}_\sigma^{(t)}(\boldsymbol{p}_\sigma^{(t)})^T - \mathbf{I}\right) + \frac{c_\mu}{2}\left(\sum_{i=1}^{\mu} w_i \boldsymbol{z}_{i:\lambda}^{(t)} (\boldsymbol{z}_{i:\lambda}^{(t)})^T - \mathbf{I}\right)\right]$        ▷ MA-ES
17:  for $i \leftarrow 1, \ldots, m$ do        ▷ LM-MA-ES
18:    $\boldsymbol{m}_i^{(t+1)} \leftarrow (1 - c_{c,i})\boldsymbol{m}_i^{(t)} + \sqrt{\mu_w c_{c,i}(2 - c_{c,i})} \sum_{j=1}^{\mu} w_j \boldsymbol{z}_{j:\lambda}^{(t)}$        ▷ LM-MA-ES
19:  $\sigma^{(t+1)} \leftarrow \sigma^{(t)} \cdot \exp\left[\frac{c_\sigma}{2}\left(\frac{\|\boldsymbol{p}_\sigma^{(t+1)}\|^2}{n} - 1\right)\right]$
20:  $t \leftarrow t + 1$
21: until stopping criterion is met
```

Figure 17. A comparison of CMA-ES, MA-ES, and LM-MA-ES[15]

that 3 trials from the neural network without an improvement to fitness would result in termination of the episode.

Ray [21] was used to manage multiple CompilerGym environments simultaneously and to unify them into a vectorized interface for the neuroevolition algorithms. Due to computational constraints in terms of core count, the neural network population was split into batches for evaluation on the environments. Each CompilerGym environment is connected to EvoSAX through a Ray remote actor which functions as a wrapper to a CompilerEnv object. The CompilerEnv object is given the population of neural networks to evaluate on the remote CompilerGym environments. The experiments were conducted on a Dell XPS-17 Laptop with a i9-12900K processor possessing 20 threads, 16GB RAM, and an NVIDIA 3060 Laptop GPU with 6GB vRAM. 8 CompilerGym environments were run concurrently, with a population size of 40 being split into 5 batches for sequential evaluation. Larger population sizes led to experiment failures due to a CompilerGym worker process dying and being restarted. Further work should improve the robustness of parallelized interaction with the CompilerGym environment. Information such as best fitness values, action distributions, and execution time were serialized and saved to disk for each of the datasets to allow for interrogation of the data without need for rerunning experiments and to also reduce pressure on demand for RAM. The rate-limiting component of the evolutionary process is the CompilerGym evaluation as compiling programs can be time consuming depending on the size.