Neuroevolutionary Compiler Control for Code Optimization

Kade Heckel
kh543@sussex.ac.uk
University of Sussex
Brighton, East Sussex, United Kingdom

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[1], 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[2] program set while demonstrating the highly adaptive properties of the neuroevolution approach.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Neural networks*; **Genetic algorithms**.

KEYWORDS

Neuroevolution, Compilers

ACM Reference Format:
Kade Heckel. 2023. Neuroevolutionary Compiler Control for Code Optimization. In *Genetic and Evolutionary Computation Conference Companion (GECCO '23 Companion), July 15–19, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583133.3596380

1 INTRODUCTION AND PRIOR WORK

The process of applying code optimizations is a complex process currently guided by heuristics. In an effort to replace the universal and non-adaptive optimization heuristics used in compilers with a more dynamic process, recent research has explored the application of reinforcement learning to different aspects of the compiler optimization process, where a neural network makes optimization decisions instead of a hard coded hueristic. For example, within the LLVM compiler[3] deep RL has been applied to the automatic vectorization of loops [4] and the in-lining of function calls [5] using algorithms such as Proximal Policy Optimization [4, 6] or Policy Gradients and Simple Evolution Strategies [5, 7, 8].

GECCO '23, July 15–19, 2023, Lisbon, PT

ACM ISBN 979-8-4007-0120-7/23/07…$15.00
https://doi.org/10.1145/3583133.3596380

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 LLVM optimization passes to a program with the reward signal being the relative code size reduction achieved compared to LLVM's default heuristics. The leading 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. [1] In this work, neuroevolution was used for the first time in the CompilerGym environment to evolve a population of multilayer perceptrons to direct the compiler optimization process.

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.[8] 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. 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.[6] Advantages of neuroevolutionary methods include not needing to perform gradient computations as well as possessing high scalability while being easy to implement, at the cost of being less sample efficient than gradient-based RL algorithms such as PPO. [5]

Three datasets were used in this work's experimentats: 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.[9] The cBench dataset which contains a larger set of 23 programs with similar in composition to CHStone was released as a benchmark suite for collective and distributed optimization of computer programs.[2] 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 datasets.[10][1]

2 METHODOLOGY

The Low Memory Matrix Adaptation Evolution Strategy [11] was employed for evolving neural network controllers utilizing the implementation provided in the EvoSAX Python library[12] which is built on top of the JAX library.[13] LM-MA-ES is an optimized version of Covariance Matrix Adaptation - Evolution Strategy which 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 initializing a population and then using recombination, mutation, and selection operators to improve solutions, EDAs evolve the parameters which describe the solution population. LM-MA-ES was selected for the experiments as it builds upon enhancements made to the CMA-ES algorithm, specifically the ability to operate on models with tens of thousands of parameters with adequate population sizes for a given hardware budget. This work presents an empirical investigation into the use of neuroevolution for compiler control and finds it competitive with gradient-based reinforcement learning methods and costly brute force searches. The Autophase[14] observation space, which describes the number of control flow and instruction counts as well as structural patterns, was used as inputs to the neural networks which then selected actions. Autophase provides a richer description of the program than raw instruction counts while also being efficient to process as an input to a neural network. 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 ReLU 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 that 3 trials from the neural network without an improvement to fitness would result in termination of the episode.

2.1 Setup

Ray [15] was used to manage multiple CompilerGym environments simultaneously and to unify them into a vectorized interface for the neuroevolution 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 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. 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.

3 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.

Figure 1 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. However, there appears to be diminishing returns as $k = 15$ resulted in minimal performance increase while still increasing the evolution time by a significant margin. The maximum fitness achieved on the cBench dataset was 1.0469 for $k = 15$, which is competitive with the 1.047 achieved by using a Graph Attention Network trained using 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.[1] Figure 2 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 respectively 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 minutes, drastically improving on the 7 hours needed to train a PPO model to assist guided search in the state-of-the-art approach. [1] The interdataset similarity between the CHStone and cBench datasets and the dissimilarity with the BLAS programs is depicted in figures 3 and 4, where the cosine similarity is 30% lower when comparing cBench to BLAS vs cBench and CHStone. These instruction count cosine similarity plots support the distinction between the first two sets and the BLAS programs in terms of general purpose versus arithmetic programs having differing characteristics. [1] Further analysis into the similarity of the actions selected by the networks for given programs demonstrate that optimizations tailored to specific programs were developed as shown in figure 5.

4 CONCLUSIONS AND FUTURE WORK

4.1 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 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.

Another line of investigation would be using a Tab Transformer, which could leverage both the normalized instruction count vectors and the instruction embedding vector observation space provided by CompilerGym[1, 16]. This approach could use ANGHABENCH

	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

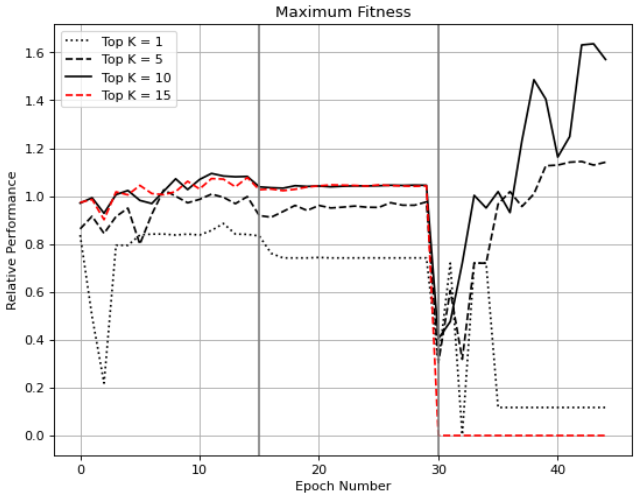

Figure 1: Best fitness attained per epoch. Vertical lines at epochs 15 and 30 denote transition from the CHStone-v0 to cBench-v1 and cBench-v1 to BLAS-v0 respectively. $k = 15$ is reported as 0 for BLAS as the experiments resulted in crashes from memory/interprocess communication errors.

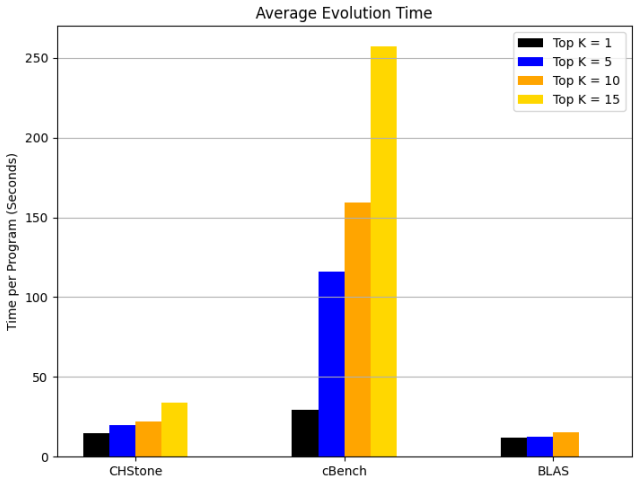

Figure 2: Average wall execution time per program to evolve for 15 epochs. No time is listed for $k = 15$ for BLAS as the experiments failed due to memory and interprocess communication issues.

as it contains over 1 million C programs scraped from public code repositories, providing a vast set of programs to train on[17]. 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[18].

Finally, learning a surrogate model for compilation could be 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 convergence to quality solutions.

4.2 Conclusions

It was found that framing the task as a batched decision problem similar to how compilers bundle together methods into optimization passes was drastically beneficial to the adaptation process. 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 selecting the actions corresponding to the top-k highest network activations it is more likely for a reduction in code size to occur, reducing the sparsity of the reward signal when executing ineffective actions. However, there exists a point of diminishing returns for the number of consecutive actions selected by a network demonstrated by the negligible improvement from $k = 10$ to $k = 15$. As evidenced by the temporal data in figure 2 and connecting it with the fitness data of figure 1, 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 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.

In conclusion, 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 efficient, achieving an average relative code size reduction of 1.047 making it competitive with gradient based approaches. The results indicate that a neuroevolutionary approach is able to rapidly adapt to the compiler optimization task and achieve high performance with behavior adapted to specific program types, and recover when the program distribution shifts to a previously unseen set as was the case with the cBench to BLAS transition.

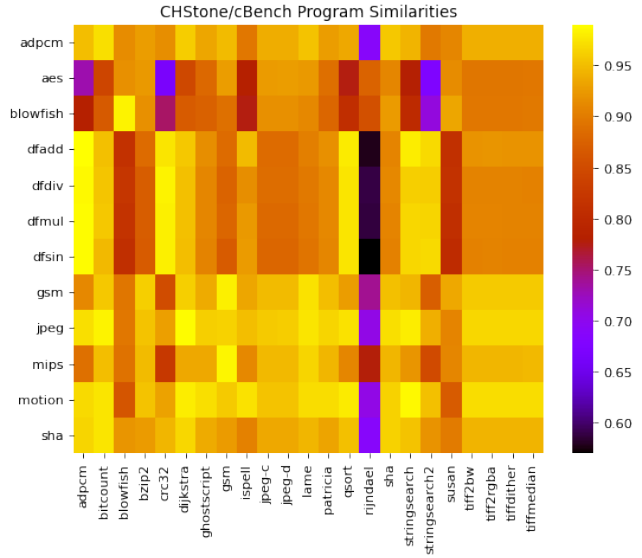

Figure 3: Similarities between the CHStone and cBench datasets; note the high similarity between the two sets with the exception of Rijndael, which is only similar to the AES and Blowfish programs.

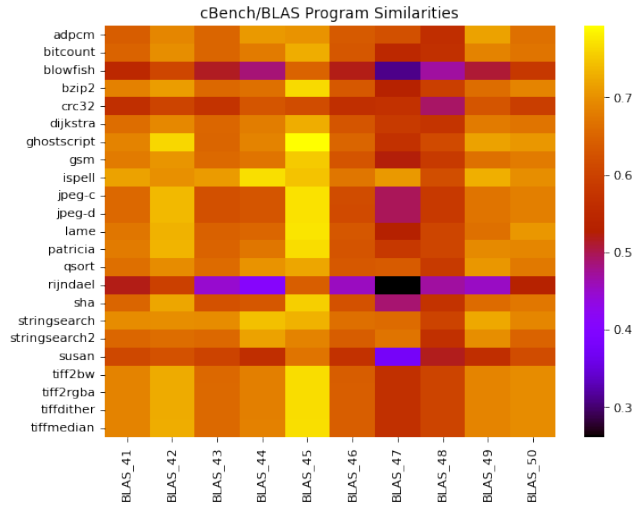

Figure 4: Cosine similarities between the cBench and BLAS datasets. Note the substantial drop in similarity compared to Figure 3.

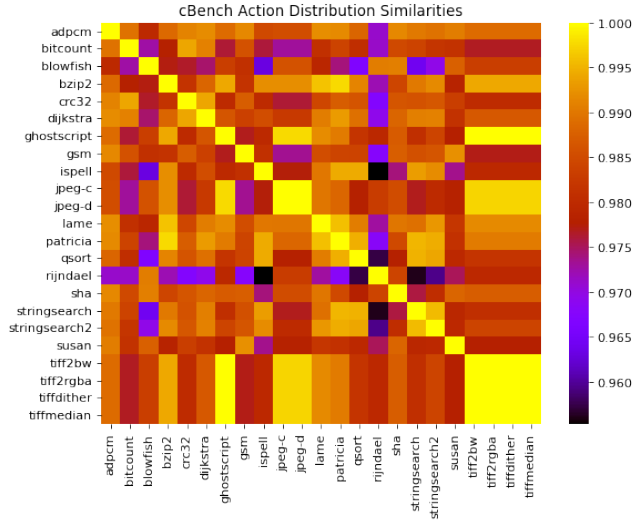

Figure 5: Cosine similarities of actions for cBench programs.

REFERENCES

[1] 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.

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

[3] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[4] 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. *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, Feb 2020.

[5] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Xinliang Li. Mlgo: a machine learning guided compiler optimizations framework. *ArXiv*, abs/2101.04808, 2021.

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

[7] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, and John Wawrzynek. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning, 2020.

[15] 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.

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

[17] 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.

[18] 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.