Target-independent XLA optimization using Reinforcement Learning

Milan Ganai[*]
University of California San Diego
mganai@ucsd.edu

Haichen Li
Amazon
lhaiche@amazon.com

Theodore Enns
Amazon
ennst@amazon.com

Yida Wang
Amazon
wangyida@amazon.com

Randy Huang
Amazon
renfu@amazon.com

Abstract

An important challenge in Machine Learning compilers like XLA is multi-pass optimization and analysis. There has been recent interest chiefly in XLA target-dependent optimization on the graph-level, subgraph-level, and kernel-level phases. We specifically focus on target-independent optimization XLA HLO pass ordering: our approach aims at finding the optimal sequence of compiler optimization passes, which is decoupled from target-dependent optimization. However, there is little domain specific study in pass ordering for XLA HLO. To this end, we propose introducing deep Reinforcement Learning (RL) based search for optimal XLA HLO pass ordering. We also propose enhancements to the deep RL algorithms to further improve optimal search performance and open the research direction for domain-specific guidance for RL. We create an XLA Gym experimentation framework as a tool to enable RL algorithms to interact with the compiler for passing optimizations and thereby train agents. Overall, in our experimentation we observe an average of 13.3% improvement in operation count reduction on a benchmark of GPT-2 training graphs and 10.4% improvement on a diverse benchmark including GPT-2, BERT, and ResNet graphs using the proposed approach over the compiler's default phase ordering.

1 Introduction

Machine Learning frameworks use Machine Learning compilers to convert neural networks into hardware readable specific code. They primarily use heuristic based approaches to solve optimizations problems at the different levels of the compiler stack. In the past several years, search-based machine learning methodologies have been proposed to optimize on the various levels of the compiler stack. Approaches have looked into sub-graph and kernel (fused operation nodes) level optimizations, optimizations of specific compiler passes, or joint optimizations across the graph, sub-graph, and kernel levels. However, the problem of graph level (for instance the High Level Operations (HLO) Intermediate Representation (IR) in XLA [1]) pass ordering has been largely optimized by heuristic based approaches.

We explore multi-pass compiler optimization on Machine Learning compilers on the graph-level. We specifically aim to algebraically optimize Machine Learning XLA graphs with target-independent HLO compiler optimization passes. That is, we need to select the sequence of XLA HLO compiler optimization passes to transform the graph to optimize a specific objective. This objective may be

[*] Work conducted during an internship at Amazon.

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on ML for Systems.

instruction count, XLA operation count, or graph-size. Our contributions to this problem domain is as follows: 1) we introduce deep Reinforcement Learning algorithms to the problem space of XLA target-independent pass ordering optimizations, 2) we propose domain-specific enhancements to the deep RL algorithms in order to further improve their performance, and 3) we demonstrate the efficacy of our approaches in comparison with default.

To enable RL algorithms to interact with the compiler and train the agents, we convert the problem into a Markov Decision process framework by creating an XLA Gym infrastructure based on OpenAI's Gym, described in Section 4. We subsequently test various deep Reinforcement Learning algorithms in Section 5.1, and we propose and test our enhancements in deep Reinforcement Learning algorithms in XLA Gym in Section 5.2.

2 Related Works

2.1 LLVM phase ordering

In LLVM [2], a closely related problem is phase ordering [3, 4] which is the problem of selecting and ordering LLVM compiler optimizations. Various techniques have been proposed in phase ordering including collaborative filtering [5], design space exploration [6], and Bayesian Networks [7]. Usage of (deep) Reinforcement Learning has been seen in various LLVM compiler optimization problems such as PolyGym [8] for polyhedral loop transformations, NeuroVectorizer [9] for single step instruction vectorization, and MLGPO [10] for inlining for size. RL-based approaches for phase ordering have been explored in Autophase [11] and CORL [12]. We refer the reader to [13] for a more in depth survey of AI based techniques for LLVM compiler optimizations.

There are also several compiler optimization research tools such as OpenTuner [14], and YaCoS [15] which are autotuning frameworks and ComPy-Learn [16] for program representation. A notable environment is CompilerGym [17] which consists of various compiler optimization problems presented using the OpenAI Gym interface including LLVM phase ordering. Overall, literature on the compiler optimization pass ordering approaches have largely focused on the LLVM environments, but Machine Learning compiler target-independent pass ordering decoupled from target-dependent optimization has not been as extensively explored.

2.2 ML compiler frameworks

Within the domain of ML compilers, there has been a growth in research in optimizations across the various levels of the compiler stack. Datasets such as Tenset [18] for tensor compilers have been produced for offline learning. Autotuning has been proposed in the subgraph and kernel levels in works such as TVM [19], AutoTVM [20], Ansor [21], FlexTensor [22], Halide [23], Chameleon [24], AdaTune [25], and Tensor Comprehension [26]. Some of these approaches are currently being utilized in production-level compilers such as the one in the AWS Neuron SDK [†] [27]. Operator-level optimizations and code generation for custom hardware accelerators has been explored in AKG [28] and Mind Mappings [29]. Our methodology operates on target-independent HLO graph level in determining the optimal pass ordering — graph/sub-graph and kernel level autotuning as well as operator-level optimizations for hardware-specific optimization is orthogonal to our work. The Value Learning approach of [30] does full graph loop optimization and is effective for a single compiler stage. Reinforcement Learning based computational graph optimization (GO) [31] jointly optimizes device placement, operator fusion, and operator scheduling does not focus on multiple target-independent passes. A survey of Deep Learning compilers can be found in [32]. Overall, we introduce Reinforcement Learning to multiple compiler pass optimization and demonstrate improvement on a target-independent level.

3 Preliminaries

3.1 XLA

The proposal for a Reinforcement Learning framework may generalize to any Machine Learning compiler framework. In this paper, we test in XLA specifically [33]. XLA is a Machine Learning

[†] AWS Neuron SDK documentation site: `https://awsdocs-neuron.readthedocs-hosted.com/`

compiler that generates code for a variety of hardware targets. The compilation process can be divided into a graph level phase, kernel level hardware lowering phase, and a low level target specific phase. In the first phase XLA uses a High Level Operation Intermediate Representation which is a graph of the tensor computations. In this target-independent step, various compiler optimization and analysis passes transform the graph into an algebraically optimized output HLO graph. The nodes composing this graph are fused operations and are known as kernels. These are brought down to the target to be converted into instructions specific to the hardware. In the process, the graph is converted into various IRs like Loop IR and Backend IR. Finally, in the hardware phase, low level target specific optimizations are are performed on the machine instructions.

3.2 Markov Decision Processes

We formulate the problem of XLA multi-pass optimization as a Markov Decision Process, which can be represented by the 5-tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$. Specifically, $\mathcal{S}$ represents the state space, which is the set of all possible values of the observable features of the graph at a given point in time. $\mathcal{A}$ is the action space that consists of 53 HLO compiler optimization passes that can be used to optimize a given graph at each step. $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function which signifies the probability $T(s'|s, a)$ that an agent at state s taking some action a will transition to state s'. This is in essence an abstraction of the compiler taking a graph with some features s and an optimization pass a and outputting a new optimized graph with features s'. $R : \mathcal{S} \rightarrow \mathbb{R}$ is a reward function that determines the immediate gain $r(s)$ of an agent being in a particular state s. Finally, the discount factor $0 << \gamma < 1$ discourages postponing good actions. A policy $\pi_\theta : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ parameterized by θ provides the probability $\pi_\theta(a|s)$ that an agent will take action a given it is in state s. The goal is to learn the optimal policy π^* that maximizes expected cumulative discounted reward, i.e. returns.

4 Gym Functionality

For simulating the Markov Decision Process for the XLA multi-pass optimization, we use the OpenAI Gym [34] structure and create the XLA Gym environments. Specifically, this requires defining and engineering the following:

State: The state space indicates the set of values the observable features of the state of the graph can take. Because there is a wide range of values that XLA Gym operation count types can take, this is primarily represented as an array indicating minimum and maximum values of each feature.

Action: Similarly, the action space defines the set of values that represent actions. In pass ordering, it is most feasible to define the action space as discrete by mapping each action to a distinct number from 0 to one less than the total number of actions.

Reward: The reward must be manually engineered in order to best optimize for guiding the reinforcement learning agent. Because we are looking to reduce the overall operation count and for ease of calculation, the reward is a function simply of the observation features. However, if we want to discourage certain types of actions or transitions, the reward function can easily generalize to become a function of the transition (i.e. $R(s, a, s')$).

Info: An information dictionary is returned at each step of Gym. This provides environment designers to conveniently provide any additional information if needed such as additional cost information.

Reset: Beginning each learning episode, it is important to reset the environment in order to bring the agent back to a starting state and clear any needed environment variables before proceeding.

Step: Once an action has been chosen by the RL agent, the environment acts as a classic decision chain by taking the action and returning the next state and reward. Furthermore, gym environments provide a boolean indicating termination of an episode and the information dictionary.

Render: Rendering allows for a readable/parsable output at any given instance of the environment.

4.1 XLA Gym environment

Using the OpenAI Gym structure, we develop XLA Gym environments suitable for various use cases. The environments are built off of the basic environment. In general, the states are arrays with elements indicating various types of XLA operation count, and the actions are whole numbers less than the total number of 53 HLO compiler optimization passes. The reward is negative of the scaled XLA operation count, though may vary across the environments. Resetting initializes a new graph without optimizations and returns the initial observable state. At each step, an action is provided, and the next state, reward, boolean for episodic termination, and info dictionary are returned. Depending on the environment, the info dictionary may contain HLO cost analysis features such as FLOP count and transcendental count. Figure 1 shows the basic usage of the XLA Gym environment.

```
import gym, xla_gym

RENDER = True
GAMMA = 0.99

env_kwargs = {'benchmark_locs':'''list of code file locations''', 'traj_limit':'''Int'''}
env = gym.make('xla-v*', **env_kwargs)

ob = env.reset()
done = False

while not done:
    action = '''sample action from policy model like model.predict(ob)'''
    ob, reward, done, info = env.step(action)
    if RENDER:
        env.render(mode='human')

env.close()

```

Figure 1: Example usage of XLA Gym's standard environment.

5 Experiments

5.1 Evaluating Deep RL Algorithms

With our XLA Gym structure, we proceed to explore various deep Reinforcement Learning Algorithms. Deep RL algorithms contain off-policy algorithms, which train a policy different from the one used to generate and collect environment data, and on-policy algorithms, which train and collect data using the same policy [35]. The two off-policy algorithms we test include Deep Q Networks (DQN) [36] and Advantage Actor Critic (A2C) [37]; the two on-policy algorithms we test include Trust Region Policy Optimization (TRPO) [38] and Proximal Policy Optimization (PPO) [39].

5.1.1 Comparison of Deep RL Algorithms

We compare the results of benchmarking the various deep Reinforcement Learning algorithms shown in Figure 2. The dataset we use for benchmarking is from a GPT-2 [40] training loop implementation maintained internally in Amazon, adapted from the NVIDIA Megatron-LM [41] project. The metric we use is the geometric mean over all the testing benchmarks of the ratio between the operation count reduction using the RL agent to that of the default HLO passes used in the AWS Neuron SDK [42]. Specifically: $\left(\prod_{b=1}^{B} \frac{(I_b - R_b)}{(I_b - D_b)}\right)^{\frac{1}{B}}$ where I is initial XLA operation count, R is count after using RL agent, D is count using default HLO passes, B is total number of benchmarks, and all counts are indexed by benchmark b. Overall, PPO performs the best with an average of 13.3% improvement in XLA operation count reduction over the default HLO passes. However, the off-policy approaches had comparatively poor performance. It is interesting to note that the work of [43] shows A2C is equivalent to PPO when certain parameters are fixed — importantly the number of update epochs in PPO must be set to 1, there should be no clipping, and the KL divergence term in the loss is removed. Therefore we hypothesize these are what make PPO perform much better than A2C. Also, note the dataset used to train the RL model is different from (no intersection with) the dataset used to test the RL model and default HLO passes.

In Figure 3 we provide the improvement in operation counts of our learning based approach over the default passes on various benchmarks. In around 97% of the testing suite benchmarks, we can see that our methodology performs at least as good as the default HLO passes method. Furthermore, our approach is able to achieve up to 27.3% total improvement in operation count reduction.

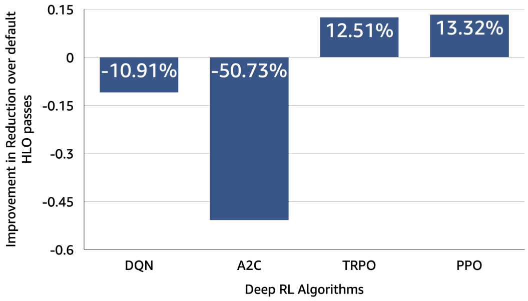

Figure 2: Comparison of various RL algorithms.

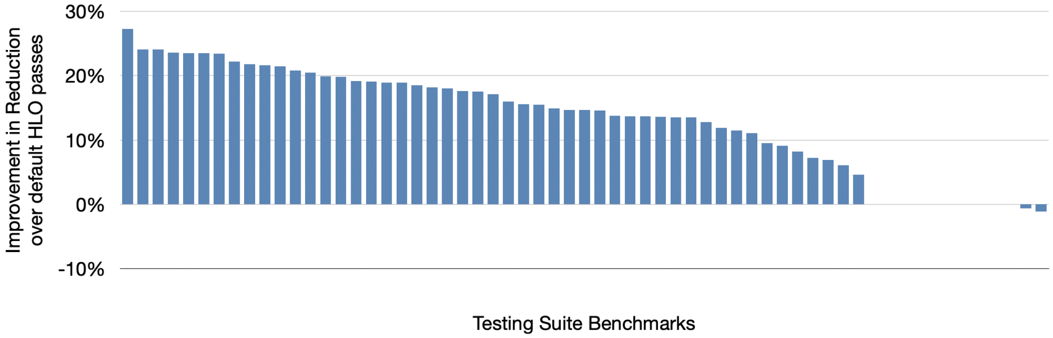

Figure 3: We present the improvement in reduction using our proposed methodology over the default HLO passes for each benchmark in out testing suite. We are able to achieve up to 27.3% total improvement. There are some benchmarks that have same performance as the default HLO as indicated by the gap on the right, likely because both approaches have reached near optimal optimization. On two benchmarks in our testing suite, we perform at most 1.2% worse than the default HLO passes.

5.1.2 Execution Time Speedup

So far we have been comparing using the HLO operation count as our metric for target-independent improvement. Using this metric, we are able to demonstrate that we can reduce our graph size in terms of the HLO operation counts using our proposed methodology. We now want to show how our proposed approach for improving XLA HLO optimization can translate to improving execution time without requiring any target-dependent optimizations. We utilize Tensorflow 2.9.2's XLA CPU compiler to generate execution time computation on a m6g.16xlarge Amazon EC2 instance on the testing suite benchmarks. Ultimately, our methodology provides an average speedup of $1.0291\times$ with standard deviation of 0.0155 (up to $1.0635\times$ speedup). The runtime speedup of $1.0291\times$ is somewhat expected because of decoupling, i.e., we are not using any target-specific cost function such as machine native instruction count for each XLA HLO operation.

5.2 Evaluating Deep RL Algorithm Enhancements

Motivated by the improvement in performance provided by deep Reinforcement Learning algorithms, particularly PPO, we seek to further boost optimal policy search by introducing domain knowledge guidance. We explore how incorporating HLO cost analysis features in various aspects of the PPO algorithm affect performance. In particular, we examine our two proposed enhancements: reward shaping and value function modification.

5.2.1 Reward Shaping

Reward Shaping introduces an artificial reward signal to the environment feedback reward [44, 45]. This must come in the form of a potential function [46]. Specifically, the work of [44] shows that function F is a potential-based shaping function if there exists a real value function $\phi : \mathcal{S} \to \mathbb{R}$ so for all $s \in \mathcal{S} \setminus \{s_0\}$, $a \in \mathcal{A}$, and $s' \in \mathcal{S}$, then $F(s, s') = \gamma\phi(s') - \phi(s)$. Furthermore, for potential-based shaping function F, they prove that every optimal policy π^* in MDP $M = \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ is also an optimal policy in MDP $M' = \langle \mathcal{S}, \mathcal{A}, T, R + F, \gamma \rangle$ and vice versa.

In this manner, we transform our initial MDP M in XLA Gym to a new one M' with provably same optimal policies by introducing a potential-based shaping function to the reward. We craft a heuristic based on HLO cost features like FLOP count and transcendental count into a function $\phi(s)$. Specifically $\phi(s) = -FLOP\_count(s) - 2 * transcendental\_count(s)$. Therefore our new reward function will be $R'(s) = R(s) + \gamma * (-FLOP\_count(s') - 2 * transcendental\_count(s')) - (-FLOP\_count(s) - 2 * transcendental\_count(s)) = R(s) + \gamma\phi(s') - \phi(s)$.

5.2.2 Value Function Modification

Another approach we propose is to introduce the cost analysis features into the value function. In deep Reinforcement Learning algorithms, the value function captures the quality of the agent in a particular state. Specifically, it predicts the returns of the agent from that state. Cost analysis features may provide a better estimate of this quality. We introduce features like FLOP count and transcendental count to the value function so the value function takes the form $V(s, f)$ where f is a vector of the additional cost analysis features.

5.2.3 Comparison of Deep RL Algorithm Enhancements

We accordingly test the enhancements to the PPO algorithm and compare it with the original PPO algorithm. The results can be seen in Figure 4. Note that in this comparison, we work with a much larger and more diverse data set of more than 300 graphs coming from models such as GPT-2, BERT [47], ResNet [48] than from evaluation in Figure 2 for better comparison of robustness of the enhancements. Overall, PPO with the shaping potential does best in comparison to plain PPO. This demonstrates that there is more room for improvement in guiding the deep RL algorithms for optimal policy search by introducing domain specific knowledge. It is also interesting that the value function enhancement has poor performance. We hypothesize this may be due to what recent papers like [49] suggest that minimum variance baselines (Value function is baseline proxy used in PPO) do not necessarily correlate to convergence to optimal policy. In essence, although introducing cost analysis features may improve the value function's quality estimate accuracy, this may in the long run backfire by potentially encouraging the agent to commit and convergence to a suboptimal policy.

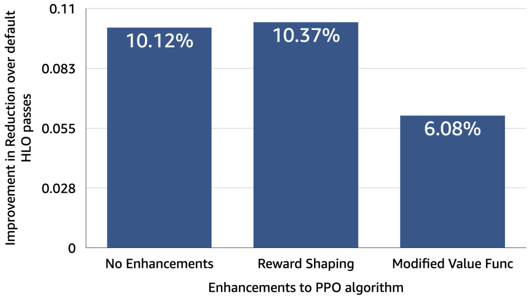

Figure 4: Comparison of the results of enhancements to PPO.

6 Conclusion

We have introduced a mostly unchartered problem of target-independent compiler optimization pass ordering in Machine Learning compilers like XLA. Specifically we propose reformulating the problem into a Markov Decision Process and address with Reinforcement Learning algorithms. We created an XLA Gym infrastructure with environments for XLA compiler optimization pass problem to specifically optimize XLA operation count, but this can be generalized to other optimization targets. Orthogonal extensions to our work include optimizing for execution speed and therefore exploring target-dependent methods like autotuning. Furthermore, we used an observation space of various types of XLA operation counts; however, there may be additional useful information in the graph structure itself. So, with much larger training suites to avoid overfitting, another future extension includes using a graph-based program representation like ProGraML [50]. To test and demonstrate the effectiveness of using RL in HLO pass ordering, we tested various off-policy and on-policy deep

Reinforcement Learning algorithms in XLA Gym. We also proposed and tested new enhancements to incorporate domain specific knowledge into the deep RL algorithms. Overall we achieve up to an average of 13.3% improvement over the default HLO passes and show further room for improvement on deep RL algorithms through domain knowledge introduction.

References

[1] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.

[2] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[3] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5):1–42, 2018.

[4] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 448–459, 2010.

[5] Stefano Cereda, Gianluca Palermo, Paolo Cremonesi, and Stefano Doni. A collaborative filtering approach for the automatic tuning of compiler optimisations. *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2020.

[6] Ricardo Nobre, Luiz Gustavo Almeida Martins, and João MP Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, 2016.

[7] Amir H. Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. Cobayn: Compiler autotuning framework using bayesian networks. *ACM Trans. Archit. Code Optim.*, 13:21:1–21:25, 2016.

[8] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. A reinforcement learning environment for polyhedral optimizations. *arXiv preprint arXiv:2104.13732*, 2021.

[9] Ameer Haj-Ali, Nesreen Ahmed, Theodore L. Willke, Sophia Shao, Krste Asanović, 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*, 2020.

[10] 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.

[11] Ameer Haj-Ali, Qijing (Jenny) Huang, John Xiang, William Moses, Krste Asanovic, John Wawrzynek, and Ion Stoica. Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 70–81, 2020.

[12] Rahim Mammadli, Ali Jannesari, and Felix A. Wolf. Static neural compiler optimization via deep reinforcement learning. *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 1–11, 2020.

[13] Hugh Leather and Chris Cummins. Machine learning in compilers: Past, present and future. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8, 2020.

[14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316, 2014.

[15] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. Yacos: a complete infrastructure to the design and exploration of code optimization sequences. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*, pages 56–63, 2020.

[16] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. Compy-learn: A toolbox for exploring machine learning representations for compilers. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–4. IEEE, 2020.

[17] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, et al. Compilergym: robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 92–105. IEEE, 2022.

[18] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[19] 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 *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[20] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[21] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[22] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[23] 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.

[24] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *ICLR*, 2020.

[25] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. *Advances in Neural Information Processing Systems*, 33:14807–14819, 2020.

[26] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *ArXiv*, abs/1802.04730, 2018.

[27] Welcome to aws neuron. `https://awsdocs-neuron.readthedocs-hosted.com/`. Accessed: 2022-09-27.

[28] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. Akg: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1233–1248, 2021.

[29] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 943–958, 2021.

[30] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. Value learning for throughput optimization of deep learning workloads. *Proceedings of Machine Learning and Systems*, 3:323–334, 2021.

[31] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Phothilimtha, Shen Wang, Anna Goldie, et al. Transferable graph optimizers for ml compilers. *Advances in Neural Information Processing Systems*, 33:13844–13855, 2020.

[32] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32:708–727, 2021.

[33] Xla: Optimizing compiler for tensorflow. `https://www.tensorflow.org/xla`. Accessed: 2022-09-07.

[34] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[37] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[38] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[40] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[42] Aws neuron sdk to optimize machine learning inference on aws inferentia chips. `https://aws.amazon.com/machine-learning/neuron/`. Accessed: 2022-09-27.

[43] Shengyi Huang, Anssi Kanervisto, Antonin Raffin, Weixun Wang, Santiago Ontañón, and Rousslan Fernand Julien Dossa. A2c is a special case of ppo. *arXiv preprint arXiv:2205.09123*, 2022.

[44] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning*, volume 99, pages 278–287, 1999.

[45] Babak Badnava and Nasser Mozayani. A new potential-based reward shaping for reinforcement learning agent. *arXiv preprint arXiv:1902.06239*, 2019.

[46] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

[47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[49] Wesley Chung, Valentin Thomas, Marlos C Machado, and Nicolas Le Roux. Beyond variance reduction: Understanding the true impact of baselines on policy optimization. In *International Conference on Machine Learning*, pages 1999–2009. PMLR, 2021.

[50] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 2244–2253. PMLR, 18–24 Jul 2021.

A Appendix

Hyperparameters and Setup	Values
On-policy parameters	
Batch size	256
Learning rate	$3e^{-4}$
Discount factor	0.99
Network Architecture	MLP $[2048, 2048]$
Entropy coefficient	0
GAE lambda	0.95
PPO clip range	0.2
Off-policy parameters	
Batch size	256
Buffer size	10^6
Learning rate	$3e^{-4}$
Discount factor	0.99
Network Architecture	MLP $[2048, 2048]$
General parameters	
Training Steps	200000
Training time	average 11 hours
Hardware	c5.18xlarge AWS EC2 Instance

Table 1: Hyperparameter Settings Details