LoopTune: Optimizing Tensor Programs with Reinforcement Learning

Anonymous Authors[1]

Abstract

Advanced compiler technology is crucial for enabling machine learning applications to run on novel hardware, but traditional compilers fail to deliver performance, and expert-optimized libraries introduce an unsustainable cost. To address this, we propose LoopTune, a deep reinforcement learning compiler that optimizes tensor computations in deep learning models for the CPU. LoopTune optimizes loop ranges and data traversal order (schedule) while using the ultra-fast lightweight code generator LoopNest to perform hardware-specific optimizations. LoopTune uses a novel graph-based representation and evaluates 5 popular RLlib algorithms to tune the model. Our results show that LoopTune generates an order of magnitude faster code than the default implementation of TVM, consistently performing at the level of the hand-tuned library Numpy.

1. Introduction

Contemporary advances in the field of machine learning (ML) have led chip designers to develop a plethora of extremely powerful chips to accommodate computationally intensive ML workloads. For instance, Nvidia introduced tensor cores (Markidis et al., 2018; Choquette et al., 2021), Intel and AMD added specialized AVX (Lomont, 2011; Jeong et al., 2012), FMA (Wittmann et al., 2015), and VNNI instructions (Tekin et al., 2021), while Google introduced TPUs (Jouppi et al., 2017). Moreover, hardware companies started making ML-specific chips such as Graphcore (Jia et al., 2019) and Cerebras (Rocki et al., 2020).

To fully harness the power of advanced hardware, advanced compiler technology is a must. However, traditional compilers have several limitations that impede their ability to do so.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

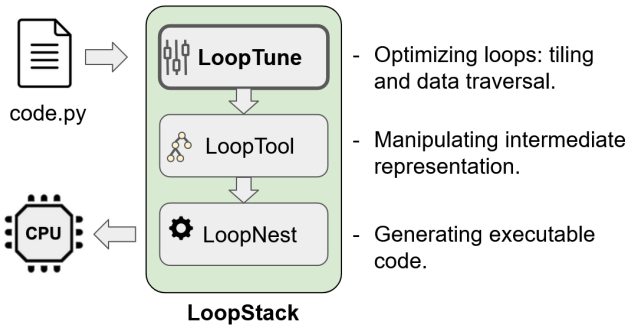

Figure 1. LoopStack architecture.

First, traditional compilers have been developed for a limited set of ISAs with similar programming models, making it difficult to adapt them to exotic hardware with different chip resources. Even with the front/back-end separation introduced by LLVM, the task remains challenging, because choices made in the representation are not easily optimized for novel hardware.

Second, as traditional compilers are extended to cover more use cases, they become increasingly complex, with hundreds of optimization passes that frequently depend on one another. This complexity leads to costly development and maintenance efforts. Finally, traditional ”catch-all” compiler techniques fail to fully utilize the novel resources available on emerging hardware designed for specific workloads.

So, what are our alternatives to the traditional compiler? Expert-optimized libraries, auto-tuners, or something else?

Expert-optimized libraries require an enormous amount of expert hours and the work must be repeated for each new device. High-performance tensor operation libraries such as cuDNN (Chetlur et al., 2014), OneDNN (Corporation, 2020b), or XNNPACK (Google, 2020) are usually tied to a narrow range of hardware devices and tend to be large in code size, which may impede their use on mobile devices.

As an alternative approach, projects like Halide (Ragan-Kelley et al., 2013) and TVM (Chen et al., 2018a) optimize a high-level tensor representation (schedule) with a discrete set of transformations such as loop reordering and tiling before compiling it to a particular target hardware with LLVM compiler. This approach provides high performance and eliminates the need for expert-optimized libraries, but

introduces an astronomical number of possible schedules.

To optimize a simpler problem, such as Local Laplacian Filters, Halide estimates a lower bound of 10^{720} possible schedules (Ragan-Kelley et al., 2013). To find performant schedules in such a huge space, Halide and TVM use genetic algorithms and parallel simulated annealing with a trained cost model (Chen et al., 2018a) respectively. Both approaches suffer from very large compilation times, making auto-tuning impractical.

Contemporary breakthroughs in deep reinforcement learning (deep RL) in complex video games, such as those in Atari (Mnih et al., 2013b), and AlphaGo (Silver et al., 2016), has inspired the compiler research communities to attempt to leverage deep RL as well. Similar to iterative algorithms, the deep RL agent explores an optimization space. However, there is one important difference - knowledge of the search space is embedded into a neural network. Inferring the neural network then replaces part of the search for optimizations.

This approach of example-driven learning, and fast optimization-space search is exactly what ML-specific, as well as general compilers, need. Recent research on RL-based compilers offers significant advantages compared to conventional techniques for many optimization problems (Haj-Ali et al., 2019; 2020; Brauckmann et al., 2021; Wang et al., 2022).

In our work, we further build on recent RL-based efforts in compiler research by extending LoopStack (Wasti et al., 2022) with LoopTune to find a performant loop schedule with deep reinforcement learning (Figure 1). By combining reinforcement learning with appropriate representations and a well-chosen optimizer, we are able to generate faster code than baseline traditional search techniques, outperform TVM's default implementation and perform at the level of an expert-optimized library Numpy.

In this paper we present the following novel contributions:

- We present LoopTune - a deep RL framework that finds performant loop ranges and schedules.
- We introduce a novel graph-based embedding of tensor computations suitable for reinforcement learning.
- We compare 5 popular algorithms from RLlib and provide the analysis of the loop schedule optimization space.

2. Background

The principal component of machine learning workloads can be expressed as a series of tensor contractions. Tensor contractions represent the generalization of matrix multiplication, trace, transpose, and other commonly used operation on matrices to higher dimensions.

Formally, we can define tensor contractions in the following way (Matthews, 2018). Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be tensors with dimensions of d_A, d_B, d_C respectively. Similar to 2D matrix multiplication, for each pair of tensors $\mathcal{AB}, \mathcal{AC}, \mathcal{BC}$ we define the dimensions both tensors will iterate together. Namely, these indices will have dimensions $I_{AB} = (d_A + d_B - d_C)/2$, $I_{AC} = (d_A + d_C - d_B)/2$ and $I_{BC} = (d_B + d_C - d_A)/2$. Then, tensor contraction can be defined with:

$$\mathcal{C}_{\Pi_C(i_0..i_{I_{AC}}, j_0..j_{I_{BC}})} = \sum_{k_0..k_{I_{AB}}} \mathcal{A}_{\Pi_A(i_0..i_{I_{AC}}, k_0..k_{I_{AB}})} \cdot \mathcal{B}_{\Pi_B(j_0..j_{I_{BC}}, k_0..k_{I_{AB}})}$$

where $\cdot$ is scalar multiplication and Π stands for all permutations of specified dimensions. Note that here we have to do all permutations to keep the result consistent, since the iterating dimensions may be chosen in any order. To simplify notation further, we can use Einstein notation and implicitly sum over dimensions that don't exist in the resulting tensor.

$$\mathcal{C}_{\Pi_C(I,J)} = \mathcal{A}_{\Pi_A(I,K)} \cdot \mathcal{B}_{\Pi_B(J,K)}$$

To allow the use of the non-linear activation function used in deep learning, we extend our notation with an element-wise operation that transforms the final result (post).

$$\mathcal{C}_{\Pi_C(I,J)} = post(\mathcal{A}_{\Pi_A(I,K)} \cdot \mathcal{B}_{\Pi_B(J,K)})$$

With these extensions, we can express not only general matrix-to-matrix multiplication (GEMM), matrix-to-vector multiplication (GEMV), and vector-to-matrix multiplication (GEVM) operations, but also general machine learning primitives such as:

- Convolutions (Krizhevsky et al., 2017) : $O_{R,C} = I_{R+K,C+J} \cdot \omega_{K,J}$
- Pooling (Krizhevsky et al., 2017) : $O_{R,C} = max(I_{2R,2C})$
- Reductions (Abdi & Williams, 2010) : $O_R = I_{R,C}$
- Transpositions (Abdi & Williams, 2010) : $O_{R,C} = I_{C,R}$
- Concatenations (Radu et al., 2018) : $O_{R,C_1+C_2} = A_{R,C_1} | B_{R,C_2}$
- Broadcast (Albooyeh et al., 2019) : $O_{R,C} = I_R$

Besides machine learning, tensor contractions are widely used in physics simulations, spectral element methods, quantum chemistry, and other fields. Despite many efforts (Grosser et al., 2012; Di Napoli et al., 2014; Matthews, 2018), none of the state-of-the-art production compilers such as GCC (Stallman et al., 1999) and LLVM (Lattner & Adve, 2004) can automatically transform naive tensor contraction loop nests to expertly-tuned implementations.

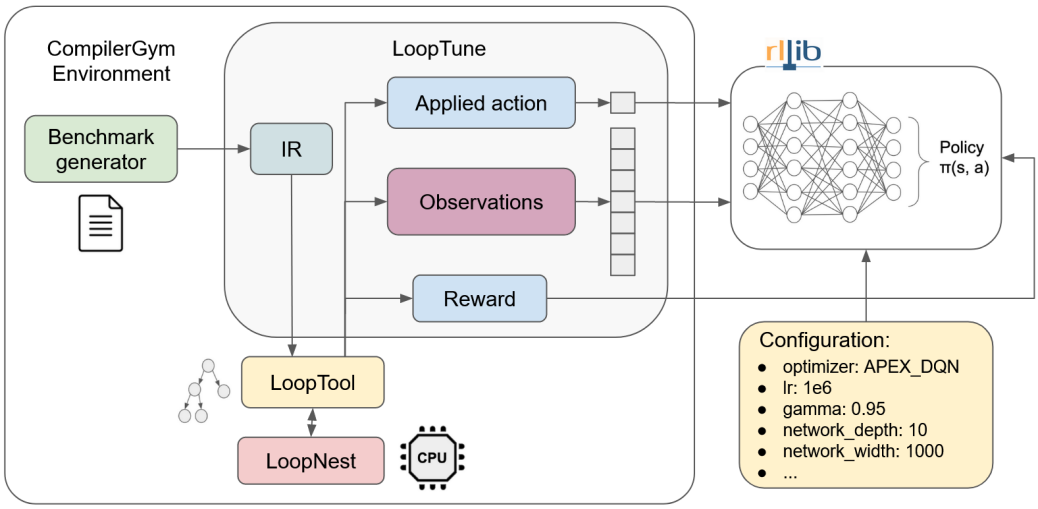

Figure 2. LoopTune training loop. LoopTune transforms generated benchmark to intermediate representation (IR), use LoopTool to apply actions, get observations, while LoopNest compile and execute loop nest providing reward. RLlib is used for training.

3. Learning to Optimize Tensor Programs

To optimize tensor operations, we separate the problem of finding optimal loop range and order (schedule) from hardware-dependent low-level optimizations, such as vectorization. To find performant schedules LoopTune uses deep reinforcement learning to train a policy network, while LoopNest applies low-level tensor optimizations and generates executable code given a schedule. We provide more details about LoopNest in Appendix A.

In Figure 2, we outline the workflow of LoopTune. The process begins by creating an environment using CompilerGym (Cummins et al., 2022). This framework allows us to map the problem of optimizing tensor computations to reinforcement learning and use state-of-the-art libraries such as RLlib (Liang et al., 2018) to train a neural network.

In each training epoch, we convert the benchmark to an intermediate representation by adding an "agent" annotation to the first loop (Figure 3). After the agent applies an action, LoopTune feeds our state representation to the reinforcement learning training loop. Finally, we compile and execute LoopTool's representation with LoopNest to calculate the action reward.

3.1. Defining an Action Space

The LoopTool interface (Wasti et al., 2022), provides LoopTune with the ability to swap the positions of two loops, given their line numbers, and split a loop, given its line number and a specified split factor. Instead of using parametric actions, which can be difficult to train, as noted by Kanervisto (2020), LoopTune defines action space as illustrated in Figure 3.

The agent uses *up* and *down* actions to move cursor without changing the loop nest structure. The *swap_up* and *swap_down* actions allow the agent to exchange the position of the current loop with its neighbor, moving the agent's cursor respectively. The *split* family of actions creates a new loop with the same iterator, dividing the loop range with the specified split parameter. If the split parameter does not evenly divide the loop range, the current loop will have a remainder or "tail", which will be executed at the end of the loop nest execution.

By limiting the action space in this way, we are able to simplify the problem in several ways. For example, a smaller number of possible actions enables the RL algorithm to explore and become more confident (Kanervisto et al., 2020) with each action for different states. This might force the agent to use longer sequences of actions to reach certain states, but this is not a problem since each action other than *up* and *down* changes the loop nest and provide non-zero reward signal. Furthermore, many states benefit from similar action sequences, which allows training to converge faster.

To keep the design simple, we decided to apply a fixed number of actions for optimization, rather than having an action that terminates the search. Our experiments have shown that having such an action often prevents exploration and converges to local minima. Instead, we rely on an implicit stop, which occurs when the agent starts oscillating between two states with the same loop nest.

3.2. Defining a Reward

For the evaluation metric, we use billions of floating-point operations per second (GFLOPS). To measure GFLOPS, LoopTune uses LoopNest to compile and execute loops on a

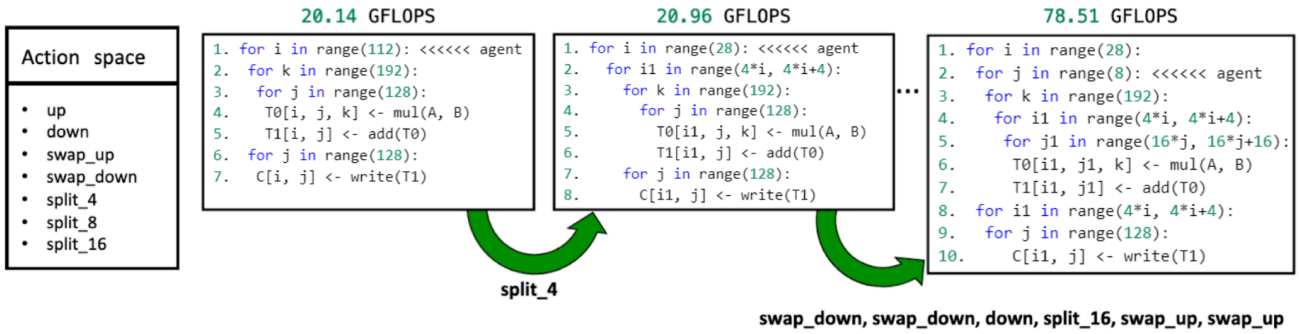

Figure 3. Optimizing ranges and order of loops for matrix multiplication using LoopTune's action space.

CPU. To ensure reliable results, LoopNest excludes the first 20 iterations as a warm-up and times multiple executions of the loop nest, taking the fastest measurement.

During training, the agent applies action (A) from state S, moving it to the next state S'. The feature extractor maps the internal representation to the vector (S) which is used as input to the neural network. LoopNest calculates the reward for the applied action using the formula:

$$reward = \frac{GFLOPS(S') - GFLOPS(S)}{GFLOPS\_PEAK\_PERFORMANCE}$$

This normalizes all rewards, making training more stable. Rather than relying on peak performance from hardware specifications that may be imprecise, we evaluate peak performance empirically. Finally, we send a tuple (S, S', A, R) to the RL library that performs one training step.

3.3. Defining the State Representation

For state representation, we use the graph shown in Figure 5. On this graph, there are 3 kinds of nodes: loop (rectangles), data (ellipses), and computation (diamonds). There are 3 kinds of edges. Black edges connect loops and computations that are nested from top to bottom. Blue edges represent data flow, while red edges represent the strides of each loop accessing each tensor. Stride is the distance in memory between two elements of a tensor when we increment only the index of a given loop. If this number is large, the iterating loop will try to fetch distant data in memory that may not be stored in the cache, which can result in a cache miss.

To make our representation usable for standard RL optimizers, we map the key features to a vector. In our vector representation, each loop is described with 20 integer values, namely:

- (1) Is the agent's cursor on the loop
- (1) Loop size
- (1) Loop tail
- (1) Does loop belong to computation or write nest
- (16) Histogram of strides frequency

The histogram of strides frequency (Figure 4) represents the cumulative distribution of access strides for each loop. In other words, it shows how many accesses with given strides are produced from the given loop. Since stride can be an arbitrary integer larger than zero, we discretize strides to bins of size 2^N, where $N \in \{0...15\}$.

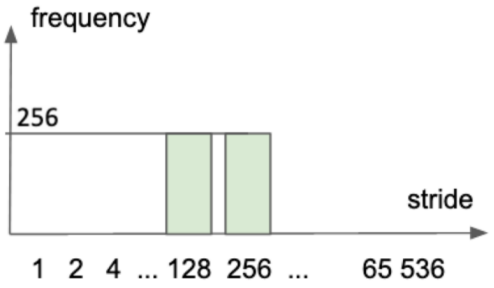

Figure 4. Histogram of strides frequency.

3.4. Library for Reinforcement Learning

To optimize the training process of our reinforcement learning model, we have chosen to use RLlib (Liang et al., 2018), the performant library for reinforcement learning. In our work, we evaluate several learning algorithms, supported by RLlib, including Deep Q Learning (DQN), Apex Deep Q Learning (APEX_DQN), Proximate Policy Optimization (PPO), Actor-Critic (A3C), and Impala.

DQN (Mnih et al., 2013a) attempts to learn the state value function by using experience replay for storing the episode steps in memory for off-policy learning, where samples are drawn from the replay memory at random.

APEX_DQN (Horgan et al., 2018) creates instances of environment for each actor and collects the resulting experience in a shared experience replay memory prioritizing the most significant data generated by actors.

PPO (Schulman et al., 2017) alternates between sampling data through the interaction with the environment while using stochastic gradient ascent with minibatch updates.

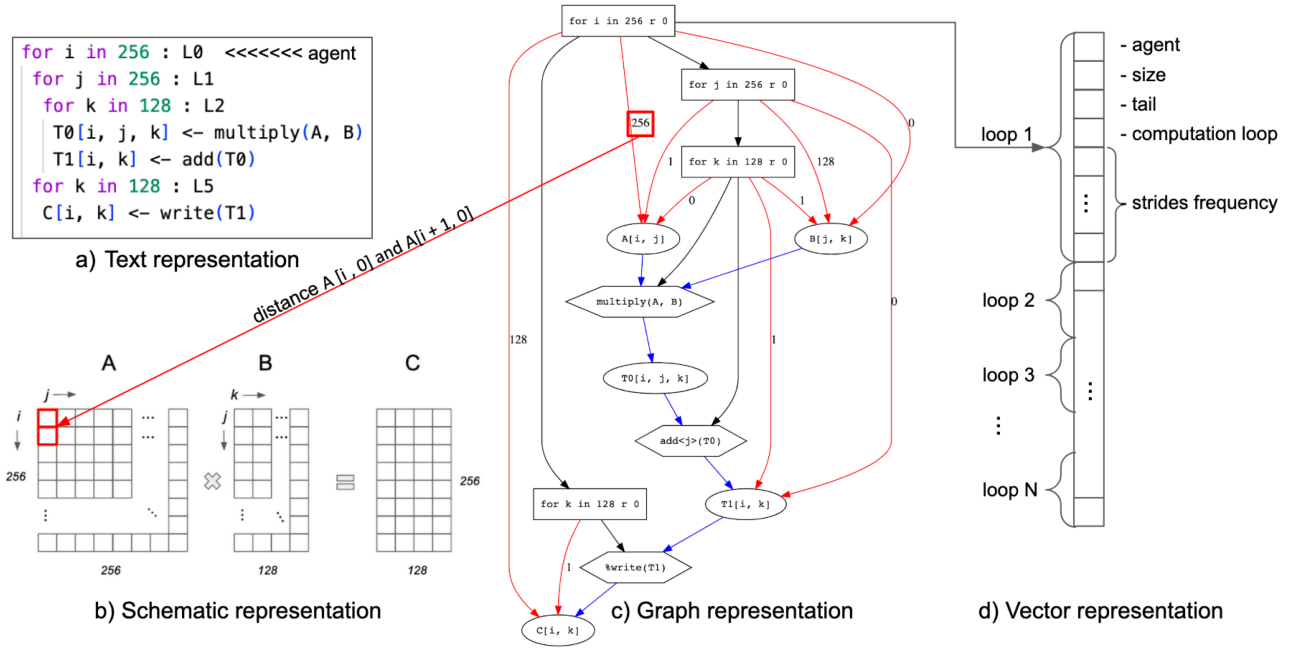

Figure 5. Text representation shows the algorithm. Schematic representation shows the memory layout. Graph representation explains nesting order (black), access pattern (red), and data flow (blue). Vector representation aggregates graph representation for the training.

A3C (Mnih et al., 2016) calculates gradients on the workers directly in each episode and only broadcasts these gradients to the central model. Once the central model is updated parameters are sent back to the workers.

IMPALA (Espeholt et al., 2018) provides a scalable solution for collecting samples from individual agents and running stochastic gradient descent in the central loop.

4. Search to Optimize Tensor Programs

The traditional approach for auto-tuning tensor programs is based on hill climbing, genetic and various search algorithms (Ashouri et al., 2018). These algorithms can find performant schedules for a single program, but the search time and the quality of the solution depend heavily on the smoothness of the optimization space. If the optimization sequence to highly rewarded states includes some actions that produce negative rewards, hill climbing algorithms can converge to local minima. Genetic algorithms, on the other hand, require a lot of time to converge and high computation resources.

We implemented the following set of search algorithms (Figure 6) to identify the difficulty of the problem:

- Greedy search with lookahead of 1 and 2
- Beam Depth First Search (BeamDFS) with width 2, 4
- Beam Breath First Search (BeamBFS) with width 2, 4
- Random search

First, we introduce the family of Greedy search algorithms with arbitrary lookahead. In each step of this algorithm, we evaluate all possible states within applying lookahead steps and apply the step toward the most promising state. With a lookahead of 1, the agent stops if there is no better action than the current state, while the lookahead of 2 enables the agent to tolerate one bad step that leads to a more promising solution. Ideally, with a large enough lookahead, we would be able to overcome the problem of local minima for actions with negative rewards. Unfortunatelly, such computation comes with the cost of $O(steps * |action\_space|^{lookahead})$, which gets prohibitevly expensive for larger lookaheads.

Second, we implemented a family of Beam search algorithms with arbitrary width. In each step, we calculate the best width actions and expand them further until we reach the specified depth of the search tree. Expansion of the states could be done in depth-first (BeamDFS) and breadth-first (BeamBFS) manner and search properties drastically differ when search time elapses before the full search graph is constructed. Complexity of both of these approach is $O(width^{steps})$, where $width < |action\_space|$.

BeamDFS can be seen as an extension to the Greedy algorithm with a lookahead of 1, with few additions. It doesn't terminate if the next state is worse than the current and it recursively visits all states of the search graph where each node has maximum *width* children. This enables it to tolerate non-convex parts of spaces as long as the optimal action ranks better than other actions in the current step.

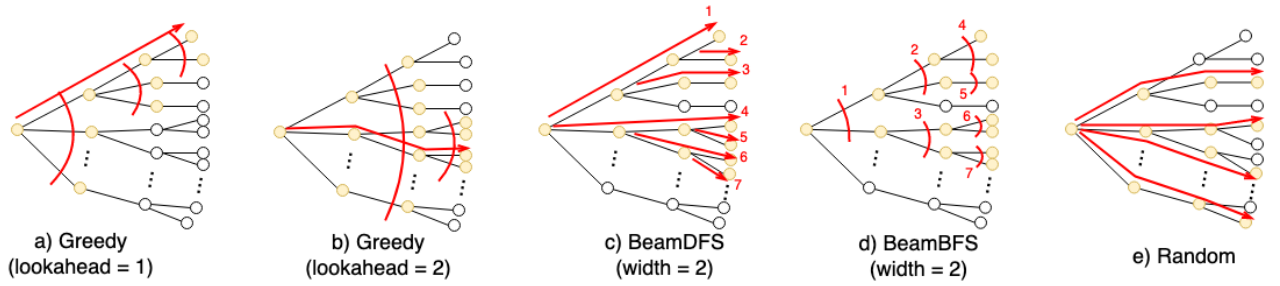

Figure 6. Traditional search approach in finding the optimal sequence. Actions (edges) are sorted by performance of the next state.

BeamBFS variant finds iteratively a performant action sequence as it builds a search graph for each number of steps. This approach would be beneficial if the performant sequence is shorter than the specified search depth.

Finally, Random search randomly chooses a sequence of actions with a specified length. The benefit of this search is that it can uniformly explore a large number of diverse states providing a general idea about the landscape. From our experiments, random search provides surprisingly good results that we elaborate on in the next section.

5. Evaluation

We evaluated LoopTune on a series of benchmarks to answer to following questions:

- How do different RL algorithms compare to each other?
- How does LoopTune compare to traditional search algorithms?
- How does LoopTune compare to optimized libraries and auto-tuners like TVM?

Benchmark dataset consists of synthesized loop nests for matrix multiplication. The matrix multiplication dataset has 2197 untiled loop nests for matrices with dimensions in the range from 64 to 256 with the step of 16.

Experiments are performed on an Intel Xeon CPU running on 2.20GHz, with 40 CPU cores and 2 Nvidia Quadro GP100 GPUs. CPU has cache sizes L1(data/instruction) 1.3 MB, L2 10MB, L3 52MB.

5.1. RLlib Training Analysis

To train LoopTune we use Ray's RLlib library (Liang et al., 2018). After we define our environment in CompilerGym and register it with RLlib, we need to instantiate the appropriate trainer and find their most promising hyperparameters. To find the best trainer we compare PPO, A3C, DQN, APEX_DQN, and Impala. In all cases as a model, we use the network with fully connected layers, with arbitrary width and the number of layers.

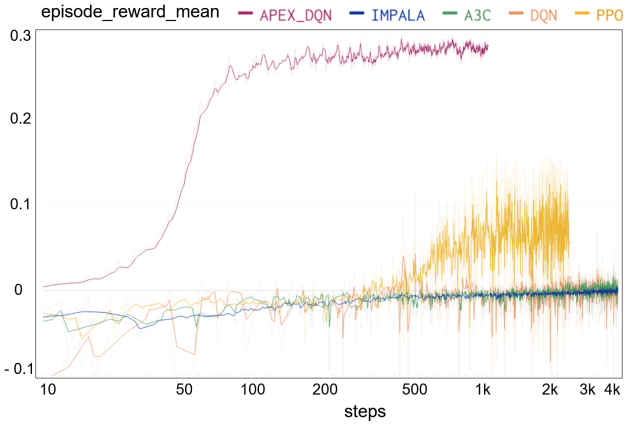

Figure 7. Average reward per epoch for RLlib algorithms during training of 4000 steps.

To find the optimal parameters for each trainer, we run a hyper-parameter sweep for the learning rate, exploration factor, depth, and width of the neural network. After finding the best parameters for each trainer, we run the final training for 4000 iterations and stop training early if the average reward per epoch converged. In each iteration, the optimizer applies the end evaluates the episode of 10 actions, and updates the neural network. Finally, we compare trainers by plotting the $episode\_reward\_mean$ which represents averaged increase of GFLOPS achieved in the episode normalized to the peak performance of the device (Figure 7).

We found that the APEX_DQN trainer performs an order of magnitude better than other trainers, converging after roughly 200 steps and providing an average increase of 30% of the peak performance (peak = 114.204 GFLOPS). In contrast, PPO required more than 1000 steps to converge to an improvement of 8% of the peak, while Impala, A3C, and DQN have not been able to achieve positive results.

We believe that the superiority of APEX_DQN lies in the capability to prioritize the most significant experiences generated by the actors. We further compare the APEX_DQN solution with non-RL approaches.

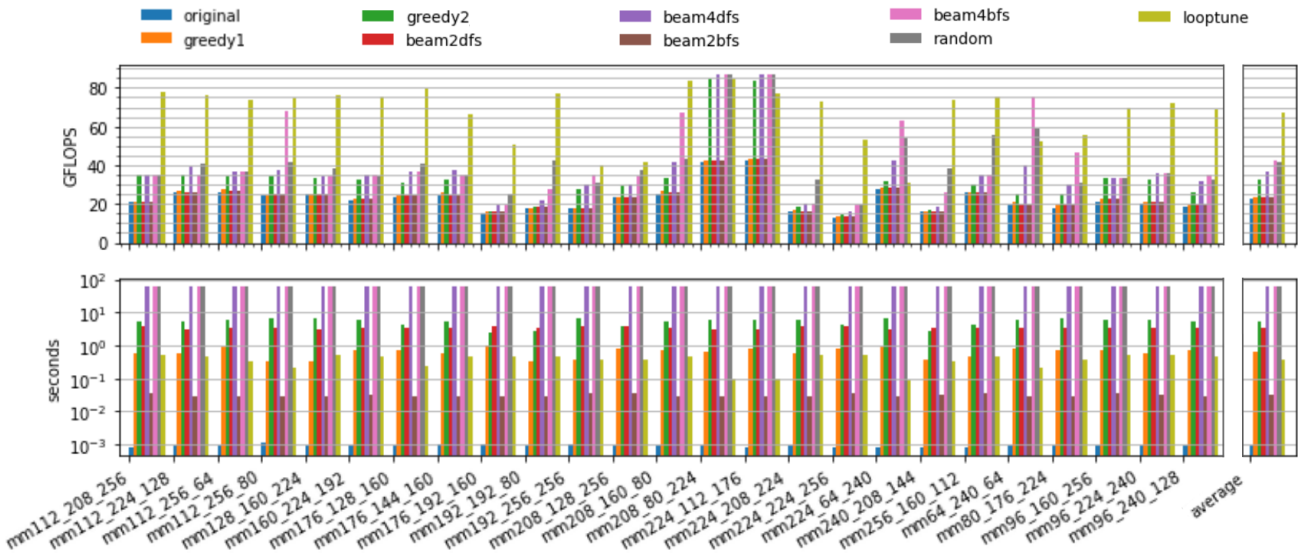

Figure 8. Achieved performance (**higher** is better) and search time (**lower** is better) of test benchmarks given 60 seconds for search.

5.2. Comparison to Search Based Approaches

To evaluate the difficulty of optimization space, we run a set of traditional search algorithms including Greedy search with lookahead of 1 and 2, BFS and DFS variant of Beam search with widths 2 and 4, and Random search. We implemented each search with caching to avoid repeating evaluations of the same states. We run each search on a test dataset of 25 benchmarks, setting the time limit to 60 seconds. To compare traditional searches to policy generated from the RL approach, we show the search time and achieved performance of the produced code on Figure 8.

In 22 out of 25 benchmarks APEX_DQN policy network outperforms the best traditional searches by 1.8x at average in less than a second, which is an order of magnitude less time. To better understand the characteristic of each search we present the speedup distribution in Figure 9.

Increasing lookahead to 2 improves Greedy search's performance. Beam2BFS and Beam2DFS achieve poor results, despite exploring the entire search subtree, which implies that performant schedules have non-performant actions. Increasing the width to 4 significantly boosts performance, outperforming Greedy2. The success of Random search further emphasizes that optimization space is non-linear. Finally, RL policy network significantly outperforms all search methods by learning useful patterns in navigating optimization space.

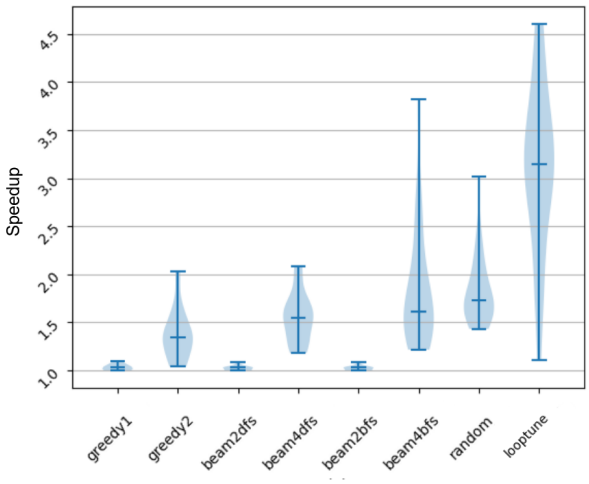

Figure 9. Speedup distribution for searches from Figure 8.

5.3. Comparison to Numpy and TVM

Next, we compare the performance of LoopTune to popular hand-tuned library for tensor operations – Numpy, and the default implementation of widely used auto-tuner – TVM (Figure 10). Numpy uses Intel's state-of-the-art MKL implementation of BLAS, while TVM applies high-level optimizations on the given schedule and uses LLVM to create an executable.

We used official documentation from TVM (documentation version, 0.11.dev0) to implement matrix multiplication for the examples from the test set. This implementation of TVM includes blocking, loop permutation, and vectorization optimizations, which are the same set of optimization we are using for LoopTune. To get the best results for our architecture we enable the *"llvm -mcpu=core-avx2"* option.

Our results demonstrate that LoopTune outperforms the default implementation of TVM by an order of magnitude on average, consistently performing at the same level as the Numpy library.

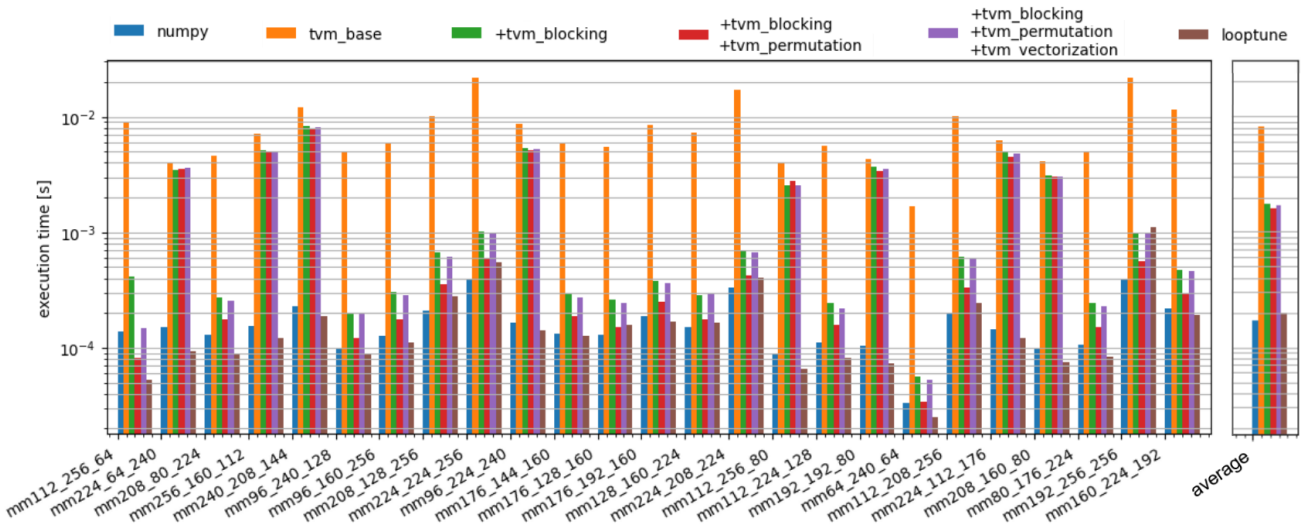

Figure 10. Execution time (**lower** is better) of test benchmarks for Numpy, TVM, and LoopTune for AVX2.

6. Related Work

Tensor-based mathematical notation, first used by APL (Abrams, 1970), is used in modern frameworks such as NumPy (Van Der Walt et al., 2011), Matlab's Tensor Toolbox (Bader & Kolda, 2006), Intel MKL (Corporation, 2020a), PyTorch (Paszke et al., 2019) and Tensorflow (Abadi et al., 2016) for manipulating tensors, performing customized operations, and executing machine learning algorithms. However, these libraries may not optimize custom hardware. Besides machine learning, tensor computations are also used in quantum chemistry simulations with libraries such as Tensor Contraction Engine(Di Napoli et al., 2014), LibTensor (Epifanovsky et al., 2013), and Cyclops Tensor (Solomonik et al., 2014).

Projects like ATLAS (Whaley & Dongarra, 1998), FFTW (Frigo & Johnson, 1998), PetaBricks (Ansel et al., 2009), OpenTuner (Ansel et al., 2014), nGraph (Cyphers et al., 2018), XLA (Sabne, 2020) , Glow (Rotem et al., 2018), and MLIR (Lattner et al., 2021) use search-based autotuning to optimize performance for custom hardware.

Halide (Ragan-Kelley et al., 2013) is the first influential work to propose the separation of computation and schedule for optimizing image processing and tensor computations. It uses a declarative language to specify tensor computations and a separate language for scheduling its execution. TVM (Chen et al., 2018a) extends Halide's compute/schedule concept with hardware intrinsics and defines new optimizations, while using parallel simulated annealing with a trained cost model. AutoTVM (Chen et al., 2018b) extends the TVM cost model by adding TreeGRU (Tai et al., 2015) that summarizes TVM's abstract syntax tree representation.

Polyhedral optimizers such as Polly (Grosser et al., 2011) use linear programming and affine transformations to optimize loops. Neurovectorizer (Haj-Ali et al., 2020) and Chameleon (Ahn et al., 2020) use deep RL to improve vectorization, and MLGO (Trofin et al., 2021) and PolyGym (Brauckmann et al., 2021) explore loop schedules with RL. CompilerGym (Cummins et al., 2022) extends this idea further, allowing users to apply RL on general code optimizations with RLlib.

7. Conclusions

We present LoopTune, an optimization tool for tensor computations that utilizes deep reinforcement learning. LoopTune selects loop ranges and schedules and then employs LoopNest to tailor the chosen schedule for the target hardware. To map this problem to reinforcement learning, LoopTune introduces a unique action space, graph-based state representation, and reward signal.

By using RLlib's APEX DQN algorithm, LoopTune outperforms traditional search algorithms by at least 1.8x on average, generating code in less than a second, while traditional approaches have a search budget of 60 seconds.

Additionally, LoopTune achieves an order of magnitude better results than the default implementation of TVM. To ensure fair comparisons, we use the same class of optimizations for both TVM and LoopTune for finding loop schedules, and for TVM, we enable a special flag for targeting hardware-specific optimizations.

Finally, LoopTune consistently performs at the same level as the expert-optimized library Numpy, significantly reducing development efforts. This finding further supports the belief that deep reinforcement learning techniques will play important role in next generation of compilers.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

Abdi, H. and Williams, L. J. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

Abrams, P. S. An apl machine. Technical report, Stanford Linear Accelerator Center, Calif., 1970.

Ahn, B. H., Pilligundla, P., Yazdanbakhsh, A., and Esmaeilzadeh, H. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743*, 2020.

Albooyeh, M., Bertolini, D., and Ravanbakhsh, S. Incidence networks for geometric deep learning. *arXiv preprint arXiv:1905.11460*, 2019.

Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. Petabricks: A language and compiler for algorithmic choice. *ACM Sigplan Notices*, 44(6):38–49, 2009.

Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., and Amarasinghe, S. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.

ARM, R. Cortex-a57 software optimization guide. *ARM*, 2016.

Ashouri, A. H., Killian, W., Cavazos, J., Palermo, G., and Silvano, C. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)*, 51(5): 1–42, 2018.

Bader, B. W. and Kolda, T. G. Algorithm 862: Matlab tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software (TOMS)*, 32(4):635–653, 2006.

Brauckmann, A., Goens, A., and Castrillon, J. A reinforcement learning environment for polyhedral optimizations. *arXiv preprint arXiv:2104.13732*, 2021.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018a. USENIX Association. ISBN 978-1-939133-08-3.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pp. 3389–3400, 2018b.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Choquette, J., Gandhi, W., Giroux, O., Stam, N., and Krashinsky, R. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

Corporation, I. Mkl developer reference. https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top.html, 2020a.

Corporation, I. Onednn. https://github.com/oneapi-src/oneDNN, 2020b.

Cummins, C., Wasti, B., Guo, J., Cui, B., Ansel, J., Gomez, S., Jain, S., Liu, J., Teytaud, O., Steiner, B., et al. Compilergym: robust, performant compiler optimization environments for ai research. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 92–105. IEEE, 2022.

Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

Di Napoli, E., Fabregat-Traver, D., Quintana-Ortí, G., and Bientinesi, P. Towards an efficient use of the blas library for multilinear tensor contractions. *Applied Mathematics and Computation*, 235:454–468, 2014.

documentation version(0.11.dev0), T. How to optimize gemm on cpu¶. URL https://tvm.apache.org/docs/how_to/optimize_operators/opt_gemm.html. [Online; accessed 28-November-2022].

Domagala, L., Rastello, F., Ponnuswany, S., and Van Amstel, D. A tiling perspective for register optimization. *arXiv preprint arXiv:1406.0582*, 2014.

Epifanovsky, E., Wormit, M., Kuś, T., Landau, A., Zuev, D., Khistyaev, K., Manohar, P., Kaliman, I., Dreuw, A., and Krylov, A. I. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations, 2013.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International conference on machine learning*, pp. 1407–1416. PMLR, 2018.

Frigo, M. and Johnson, S. G. Fftw: An adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, volume 3, pp. 1381–1384. IEEE, 1998.

Google. Xnnpack. https://github.com/google/XNNPACK, 2020.

Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A., and Pouchet, L.-N. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, pp. 1, 2011.

Grosser, T., Groesslinger, A., and Lengauer, C. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

Haj-Ali, A., Huang, Q., Moses, W., Xiang, J., Stoica, I., Asanovic, K., and Wawrzynek, J. Autophase: Compiler phase-ordering for high level synthesis with deep reinforcement learning. *arXiv preprint arXiv:1901.04615*, 2019.

Haj-Ali, A., Ahmed, N. K., Willke, T., Shao, Y. S., Asanovic, K., and Stoica, I. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pp. 242–255, 2020.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., and Silver, D. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*, 2018.

Intel, R. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, Sept*, 2014.

Jeong, H., Kim, S., Lee, W., and Myung, S.-H. Performance of sse and avx instruction sets. *arXiv preprint arXiv:1211.0820*, 2012.

Jia, Z., Tillman, B., Maggioni, M., and Scarpazza, D. P. Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*, 2019.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

Kanervisto, A., Scheller, C., and Hautamäki, V. Action space shaping in deep reinforcement learning. In *2020 IEEE Conference on Games (CoG)*, pp. 479–486. IEEE, 2020.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Lattner, C. and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., and Stoica, I. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pp. 3053–3062. PMLR, 2018.

Lomont, C. Introduction to intel advanced vector extensions. *Intel white paper*, 23, 2011.

Markidis, S., Der Chien, S. W., Laure, E., Peng, I. B., and Vetter, J. S. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

Matthews, D. A. High-performance tensor contraction without transposition. *SIAM Journal on Scientific Computing*, 40(1):C1–C24, 2018.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013a.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013b.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Radu, V., Tong, C., Bhattacharya, S., Lane, N. D., Mascolo, C., Marina, M. K., and Kawsar, F. Multimodal deep learning for activity and context recognition. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(4):1–27, 2018.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6): 519–530, 2013.

Rocki, K., Van Essendelft, D., Sharapov, I., Schreiber, R., Morrison, M., Kibardin, V., Portnoy, A., Dietiker, J. F., Syamlal, M., and James, M. Fast stencil-code computation on a wafer-scale processor. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2020.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Sabne, A. Xla : Compiling machine learning for peak performance, 2020.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

Solomonik, E., Matthews, D., Hammond, J. R., Stanton, J. F., and Demmel, J. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

Stallman, R. M. et al. *Using and porting the GNU compiler collection*, volume 86. Free Software Foundation, 1999.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Tekin, A., Tuncer Durak, A., Piechurski, C., Kaliszan, D., Aylin Sungur, F., Robertsén, F., and Gschwandtner, P. State-of-the-art and trends for computing and interconnect network solutions for hpc and ai. *Partnership for Advanced Computing in Europe, Available online at www. praceri. eu*, 2021.

Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., and Li, D. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

Van Der Walt, S., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *Computing in science & engineering*, 13(2):22–30, 2011.

Wang, H., Tang, Z., Zhang, C., Zhao, J., Cummins, C., Leather, H., and Wang, Z. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pp. 129–143, 2022.

Wasti, B., Cambronero, J. P., Steiner, B., Leather, H., and Zlateski, A. Loopstack: a lightweight tensor algebra compiler stack. *arXiv preprint arXiv:2205.00618*, 2022.

Whaley, R. C. and Dongarra, J. J. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pp. 38–38. IEEE, 1998.

Wittmann, M., Zeiser, T., Hager, G., and Wellein, G. Short note on costs of floating point operations on current x86-64 architectures: Denormals, overflow, underflow, and division by zero. *arXiv preprint arXiv:1506.03997*, 2015.

A. LoopNest Backend Optimizer

LoopNest (Wasti et al., 2022) is a powerful, domain-specific compiler that is specifically designed to optimize tensor programs. It utilizes a small set of expert-designed HPC optimizations, including custom primitives in code generation, custom assembly codes, instruction reordering, r-sum, and other optimizations suggested by optimization manuals for the target hardware (ARM, 2016; Intel, 2014).

Unlike traditional compilers, LoopNest takes into account user-defined orders of operations. This simplifies the code generator design and provides a more direct mapping between the quality of the user-defined order and its performance. This feature is particularly important for finding the best sequence of actions with reinforcement learning, and we firmly believe it makes the optimization space smoother and enables faster convergence.

Furthermore, LoopNest performs loop unrolling in a way that is consistent with hardware requirements, and automatically vectorizes the innermost loop. It also applies register tiling (Domagala et al., 2014), keeping a portion of the output tensor in registers at all times. To reduce pressure on load/store units, LoopNest never generates code that spills registers to the stack, unlike traditional compilers like LLVM and GCC. It achieves this by finding the largest scope in which any modified values can fit in the register file.

When compared to LLVM (Lattner & Adve, 2004), which is commonly used as a backend choice for tensor compilers such as Halide (Ragan-Kelley et al., 2013) and TVM (Chen et al., 2018a), LoopNest achieves orders of magnitude faster compilation times while generating code that is equal or higher in performance. This claim has been validated by Wasti et. al. (2022), who provided a pair-wise comparison of both systems by measuring average compile time and execution time of generated code for the top 5 schedules for each benchmark on AMD (AVX2) architecture (Table 1). Halide is used to emit schedules for LLVM code generation, turning off runtime assertions and bound checks.

Beyond the AMD (AVX2) architecture, LoopNest achieves similar results for Intel (AVX512), Cortex A57, Cortex A73, NVIDIA Denver2, and Apple M1 architectures. Additionally, LoopNest has a small binary footprint of $250Kb$, compared to LLVM's $350Mb$ footprint, which makes LoopNest a compiler of choice for use on mobile and embedded systems.

Table 1. LoopNest vs. LLVM performance on AMD (AVX2) architecture (Wasti et al., 2022).

	Compilation time [seconds]			Execution performance [GFLOPS]		
	LLVM	LN	Ratio	LLVM	LN	Ratio
CONV-1	820.78	**2.5153**	**326.31**	86.767	**87.638**	**1.01**
CONV-2	1590.1	**11.717**	**135.7**	45.765	**71.482**	**1.5619**
CONV-3	762.52	**5.9325**	**128.53**	9.4946	**72.433**	**7.6289**
CONV-4	885.74	**41.163**	**21.518**	3.307	**90.722**	**27.434**
DWCONV-1	838.46	**0.29416**	**2850.3**	48.695	**62.541**	**1.2844**
DWCONV-2	1033.8	**0.47162**	**2192**	39.203	**53.465**	**1.3638**
DWCONV-3	969.88	**0.30031**	**3229.6**	62.873	**84.848**	**1.3495**
DWCONV-4	1000.7	**0.33429**	**2993.4**	77.071	**84.21**	**1.0926**
MM-64	697.37	**0.38452**	**1813.6**	85.808	**102.2**	**1.191**
MM-128	925.47	**1.578**	**586.47**	92.692	**102.5**	**1.1058**
MM-256	1118.5	**2.6925**	**415.41**	92.862	**100.21**	**1.0791**
MM-512	1262.3	**4.3405**	**290.83**	90.189	**98.199**	**1.0888**

B. Analysis of LoopTune optimization space

In almost all cases APEX_DQN policy network provides better results than traditional searches in less than a second, which is an order of magnitude less time. To understand more the process of search we plot how much GFLOPS different searches achieve with each action, and how much time it took to construct each layer of search graph (Figure 11).

Greedy1 terminates quickly, creating a search graph of depth 2 and being stuck to the local minima. Greedy2 can expand the graph up to depth 6, avoiding one step local minima and achieving better performance, but still exploring only small number of states.

Beam2DFS expands the graph in-depth and each layer is updated during graph construction, keeping the time curve relatively flat. BeamBFS, on the other hand, builds the search space layer by layer completing lower layers first. The fact that Beam2DFS and Beam2BFS finished before the deadline (60s) means that they constructed the whole search graph of spawn 2. Neither of two searches found a performant solution indicates that all performant schedules consists of at least one action that is best 2 actions.

Beam4DFS and Beam4BFS both terminated with a deadline which means that they only partially constructed their search graph of spawn 4. Beam4DFS's search graph includes solutions with long sequences up to 10 steps, while Beam4BFS completely explored all solutions with 5 steps. In both cases the best solutions contain long sequences of actions with non-monothonicaly increasing performance, that enables these searches to see further than Greedy search.

Random search uses all time to expand the search graph from root to depth 10 without following any metric and evaluating each state in the graph. This way Random search can uniformly explore optimization space, including sequences of non-monothonic actions.

RL policy network is able to outperform all previous algorithm by learning optimization patterns that maximize future reward. Their solution tolerates long sequences of

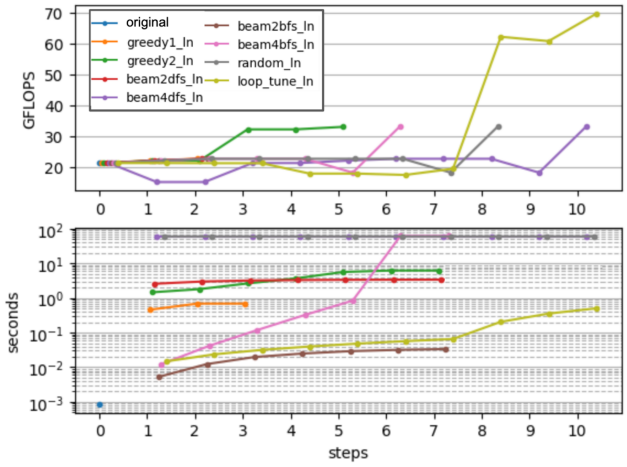

Figure 11. GFLOPS of per action and time needed for creating a search graph.

non-performant actions, being worse than all other searches from 4th to 7th step to reach performant state at 8th step. Additionally, RL policy network search time grows linearly in the length of an action sequence, which enables us to use the policy network on harder problems that require a larger number of steps. These capabilities are paramount for auto-tuning general compilers such as LLVM.