PerfDojo: Automated ML Library Generation for Heterogeneous Architectures

Andrei Ivanov
ETH Zurich
Zurich, Switzerland
andrei.ivanov@inf.ethz.ch

Siyuan Shen
ETH Zurich
Zurich, Switzerland

Gioele Gottardo
ETH Zurich
Zurich, Switzerland

Marcin Chrapek
ETH Zurich
Zurich, Switzerland

Afif Boudaoud
ETH Zurich
Zurich, Switzerland

Timo Schneider
ETH Zurich
Zurich, Switzerland

Luca Benini
ETH Zurich
Zurich, Switzerland

Torsten Hoefler
ETH Zurich
Zurich, Switzerland

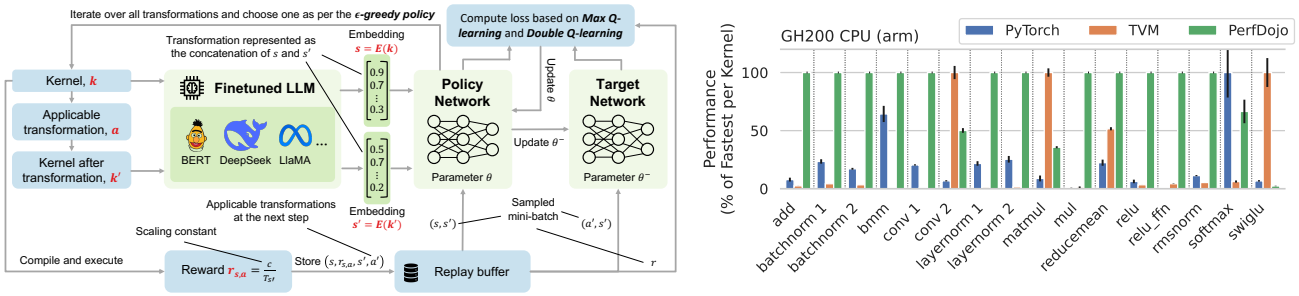


Figure 1: (a) Overview of the PerfLLM training pipeline. (b) On GH200, the geometric mean speedup of PerfDojo is 6.65× relative to PyTorch [29] and 13.65× relative to TVM [7]. The figure displays multiple variants for some kernels, reflecting the consideration of different input tensor shapes (Table 3).

Abstract

The increasing complexity of machine learning models and the proliferation of diverse hardware architectures (CPUs, GPUs, accelerators) make achieving optimal performance a significant challenge. Heterogeneity in instruction sets, specialized kernel requirements for different data types and model features (e.g., sparsity, quantization), and architecture-specific optimizations complicate performance tuning. Manual optimization is resource-intensive, while existing automatic approaches often rely on complex hardware-specific heuristics and uninterpretable intermediate representations, hindering performance portability. We introduce PerfLLM, a novel automatic optimization methodology leveraging Large Language Models (LLMs) and Reinforcement Learning (RL). Central to this is PerfDojo, an environment framing optimization as an RL game using a human-readable, mathematically-inspired code representation that guarantees semantic validity through transformations. This allows effective optimization without prior hardware knowledge, facilitating both human analysis and RL agent training. We demonstrate PerfLLM's ability to achieve significant performance gains across diverse CPU (x86, Arm, RISC-V) and GPU architectures.

CCS Concepts

• **Software and its engineering** → *Correctness*; **Software performance**; **Compilers**; **Domain specific languages**; • **Computing methodologies** → Game tree search; *Reinforcement learning*; *Natural language processing*.

Keywords

LLM, RL, HPC, ML, Compilers, Optimization

ACM Reference Format:
Andrei Ivanov, Siyuan Shen, Gioele Gottardo, Marcin Chrapek, Afif Boudaoud, Timo Schneider, Luca Benini, and Torsten Hoefler. 2025. PerfDojo: Automated ML Library Generation for Heterogeneous Architectures. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3712285.3759900

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759900

1 Introduction

The growing size [51] of machine learning models is increasing computational demands and driving the development of a wide range

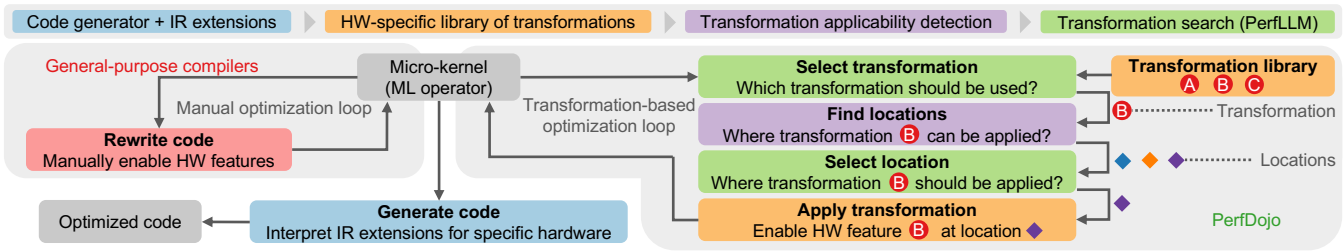


Figure 2: Manual vs. PerfDojo's transformation-centric optimization workflows.

of model architectures, high-performance accelerators (NVIDIA A100 [9], Google TPU v4 [22], Xilinx Versal FPGAs [16], Snitch [58]), and processor architectures (ARM [53], RISC-V [23]). Fully utilizing the performance of these systems is becoming increasingly difficult.

Achieving optimal performance utilization is complex due to the heterogeneity of accelerator architectures and their instruction sets. Each arithmetic primitive within machine learning necessitates specialized kernels for efficient computation. These kernels must be tailored to specific data types, contingent on the underlying model. Furthermore, models employing quantization or sparsity require implementation adjustments to accommodate the algorithmic subtleties of these features [14]. In certain scenarios, such as those leveraging flash attention, the selection of specific models and accelerators may enable operator fusion, necessitating bespoke kernel implementations [34].

Despite variations in implementation, common optimization strategies, such as tiling, vectorization, padding, and the utilization of specialized instructions, remain prevalent. However, these optimizations necessitate architecture-specific parameterization, accounting for memory and cache sizes, available specialized instructions, and other hardware attributes. Moreover, the code syntax, exemplified by CUDA [25], is vendor-dependent and dictated by available compilation stacks.

Optimizing for all these architectures is challenging. To address them efficiently, we propose a novel automatic optimization approach relying on LLMs. Central to enabling our method is PerfDojo, an environment that frames code optimization as an RL [39] game. At its core lies a flexible, human-readable representation of programs and their transformations, closely resembling mathematical formulas accompanied by performance annotations that clarify their mapping onto hardware features. Inspired by recent LLM architectures [4, 19], PerfDojo is designed with modularity and interpretability at its heart. This human-oriented design not only facilitates manual optimization but also enables RL agents to explore and apply code transformations more effectively.

Manual optimization is expensive, demanding significant man-hours to acquire the necessary architectural expertise. Even with tools assisting in the optimization process (e.g. DaCe [3]), achieving effective results remains challenging. These tools often require a deep understanding of hardware constraints to select appropriate optimizations. For instance, choosing the correct loop unrolling factors or memory access patterns still necessitates a thorough grasp of the target architecture, despite tool assistance.

Existing automatic optimization frameworks often fail to match the performance of hand-tuned libraries due to several key limitations. First, they rely on hardware-aware heuristics to guide optimization searches. This requires vendors to implement complex heuristics that represent specific hardware knowledge. Second, these frameworks impose a significant burden on vendors of new hardware architectures and framework developers, who must maintain specialized code generator backends. Finally, these frameworks lack an API for automating the fine-grained code transformation workflow that human engineers use during manual code optimization.

In this work, we propose PerfDojo (Figure 2), a code representation and environment for code transformation prioritizing the validity of code transformations. This approach ensures that the automatic exploration of the program optimization space is restricted to programs with unaltered semantics.

- Specifically, PerfDojo allows fully automated code optimization without explicit specification of hardware models or heuristics, and it provides guarantees on program semantic preservation necessary for the full automation of the code optimization process.
- Moreover, the representation is engineered for human readability and interpretability. This is not merely a convenience, but a strategic necessity. Human understanding facilitates effective debugging, iterative refinement, and the development of intuitive heuristics, all of which are essential for constructing a robust learning environment. The ability of humans to effectively interact with and understand the representation directly translates to the potential for RL agents to learn and optimize within it.
- Finally, PerfDojo shifts the paradigm from providing hardware-aware libraries to providing hardware-aware transformations, enabling vendors to extend the automatic library generation pipeline.

Our key contributions involve:

- Formulating PerfDojo: A novel representation guaranteeing semantic validity preservation throughout automatic code transformations (Section 2).
- Formulating PerfLLM: A novel automatic program optimization methodology leveraging LLMs and RL, optimizing code without prior architecture knowledge (Section 3).
- Evaluating achievable performance gains with our representation and optimization methodology on a set of ML kernels and hardware architectures (Section 4).

2 PerfDojo: A Game for Finding High-Performance Kernels

Creating a Dojo for RL agents is not trivial. Firstly, it needs to support implementations that could be handcrafted by human developers. Secondly, there should be streamlined progression toward these implementations.

One feature of the representation we require is support for manual optimization processes that can be performed by a human engineer. General-purpose compilers, such as Clang and GCC, lack fine-grained control over the application of transformations to user-specified sections of the intermediate representation. This limitation restricts the search space to cases predefined by heuristics built into the compilers, and it complicates the exposure of transformation-centric optimization pipelines to search methods that are not integrated into the compilers.

Some frameworks may allow for detailed specification of transformation schedules without checking correctness of these, making it the responsibility of a user to make sure that requested transformations are not breaking code semantics. While this may be acceptable for manual optimization, the lack of thorough applicability checks accompanied with transformations make search space of program optimizations filled with many broken implementations. Although this can be remedied with numerical verification, it will pollute the search space requiring longer exploration time by automatic optimization methods. We expect that enforcing semantic preservation in the IR manually will help RL agents to focus only on the program optimization task, instead of trying additionally to learn how to avoid semantically incorrect transformations.

A feature helpful for maintaining interpretability of IR is keeping each transformation simple, that is, atomically doing only one specific change at once. This can ensure that programmers can verify the result of this change manually, making the transformation semantics also understandable for RL agents.

Simple transformations can help formulate straightforward applicability constraints for the transformation under which semantic preservation is guaranteed, which is very challenging for non-atomic transformations, as demonstrated in FuzzyFlow [31]. In some frameworks, loop vectorization can be an example of a complex transformation, when it handleds loop tiling to the vector size, followed by unrolling of the innermost loop and attempting to replace each vectorizable operand and instruction in its unrolled body with appropriate vector counterpart. In contrast, PerfDojo requires tiling transformation to be applied explicitly, only after which vectorization can be applied under the constraint that the number of iterations in the affected loop are equal to the vector size and this loop wraps just a single instruction with vectorizable arguments.

When the program optimization search space is getting too large, existing frameworks usually resort to incorporating heuristics that guide transformations according to expert knowledge. Collecting such knowledge and describing it in a heuristic may be a very cumbersome process, limiting the speed with which these frameworks are adjusted to the new hardware features.

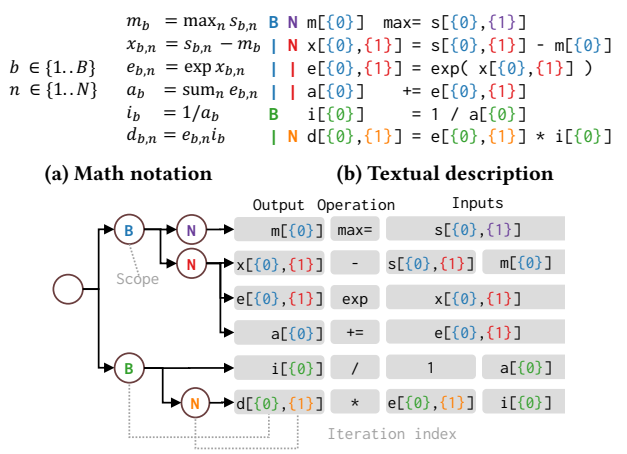
Sometimes, for the sake of simplicity, search space may be explicitly constrained beyond constraints imposed by representation (e.g. to search only over tile sizes) either because it is simpler to ensure semantic preservation for a limited subset of programs or to constrain search space with heuristics to avoid long transformation search times. Such constraints may disable finding of the most efficient implementations.

We also ask for transformations to be non-destructive, meaning they do not lose any details of the initial computation specification. For example, if loop unrolling decomposes each loop iteration to individual instruction, it may be impossible to undo it. Both human engineers and RL agents may become aware of incorrectly applied unrolling in the earlier transformation step and may want to undo it, maintaining all other transformations applied since then in place.

In Table 1, we list frameworks along with the requirements their representations satisfy. In the following sections, we describe a representation and transformation space designed to satisfy these requirements.

Table 1: Features Available in Representations of Existing Frameworks

	GCC	Polly	Halide	Dace	TVM	PerfDojo
Manual transformations	×	×	✓	✓	✓	✓
Semantic preservation	✓	✓	×	×	✓	✓
Atomic transformations	×	×	×	×	✓	✓
Heuristics not required	×	×	✓	✓	×	✓
Unconstrained search space	×	✓	×	✓	×	✓
Non-destructive transformations	×	✓	×	×	×	✓




```
for (b=0;b<B;b++) {
  m[b] = -DBL_MAX; for (n=0;n<N;n++) { m[b] = max(m[b],s[b*N+n]); }
  a[b] = 0;        for (n=0;n<N;n++) { x[b*N+n] = s[b*N+n]-m[b];
                                       e[b*N+n] = exp(x[b*N+n]);
                                       a[b]    += e[b*N+n];      }}
for (b=0;b<B;b++) {                    i[b]     = 1 / a[b];
                   for (n=0;n<N;n++) { d[b*N+n] = e[b*N+n]*i[b]; }}
```

(d) Generated code

Figure 3: Softmax kernel representations.

Figure 4: Optimization of a softmax kernel through a sequence of transformations (moves) on a CPU with AVX-512 extensions. Each move in the PerfDojo game maintains the initial program semantics.

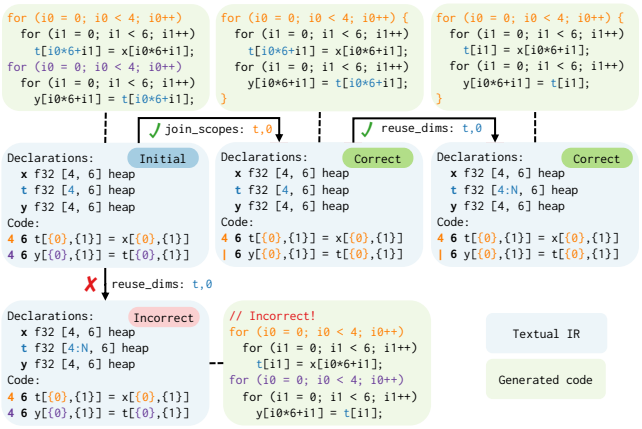


Figure 5: Program transformation example: Buffer dimension reuse (`reuse_dims`) is correctly applied with prior loop fusion (`join_scopes`), as shown in the top, but yields incorrect computation without it, as shown in the bottom.

2.1 Representation

The main data structure of our IR (Figure 3) is an ordered tree. Internal vertices, except the root node, represent single-dimensional iteration *scopes*. Leaves represent *operations*, together with associated output and inputs. The order of child vertices defines the order of their execution within the parent. Each input and output within the leaf represents a scalar value, encoded with the name of the multidimensional array and the multidimensional index within such array. Such an index can refer to the iteration over a particular ancestor scope by specifying an integer in curly braces that indicates the depth of the scope.

To facilitate operations on the graph intermediate representation (Figure 3c), we map it to a human-readable textual format (Figure 3b). In this format, vertical bars | denote a child node relationship with the nearest preceding line that does not contain a bar in the same position, indicating the parent node.

To specify the memory mapping of multidimensional arrays used in the textual description of the kernel, we include a list of buffer declarations. Each declaration specifies the buffer's name, the data type it holds, its shape as a list of dimensions (`[dim1, dim2]`), and its memory location (e.g., heap or stack), followed by an optional list of array names when multiple arrays reside in the same memory buffer. A single buffer declaration adheres to the following format:

```
buffer_name data_type shape location -> list_of_array_names
```

Each dimension can have an optional `:N` suffix, which prevents its materialization and reduces the buffer's memory footprint when the iteration order allows for memory reuse. When the list of array names is omitted, it is assumed that the buffer holds a single array of the same name.

The scope sizes in the textual description support a range of suffixes that specify the way of instantiating the iteration range: `:u` to unroll the scope, `:p` to parallelize it, and `:v` to vectorize. For kernels optimized for execution on GPUs, the suffixes `:g`, `:b`, and `:w` represent scope mapping to GPU grid, block, and warp, respectively.

In Table 2, we demonstrate various operation types, or features, that add expressiveness to the IR. Some of these features, such as *broadcast*, *constant as value*, *index as value*, and *reduction*, are mutually orthogonal. Importantly, certain features can be expressed

Table 2: Supported representation features in PerfDojo.

	Operation type	Example in textual form
Supported	Element-wise	`N M z[{0},{1}]=x[{0},{1}]*y[{0},{1}]`
	Broadcast	`N M z[{0},{1}]=x[{0}]`
	Constant as value	`N M z[{0},{1}]=x[{0},{1}]*C`
	Index as value	`N M z[{0},{1}]=x[{0},{1}]*{0}`
	Reduction	`N M z[{0}]+=x[{0},{1}]`
	Expression as location	`N M z[{0},{1}]=x[{0}*N+{1}]`
Excluded	Indirection	`N z[{0}]=x[y[{0}]]`
	Data-dependent range	`N M[{0}] z[{0},{1}]=x[{0},{1}]`
	Dependent iteration	`N z[{0}]=z[{0}-1]*y[{0}]`
	General control flow	`while z[{0}++] z[{0}]=x[{0}]*y[{0}]`

by combining others; for example, *indirection* along with *index as value* can represent *expression as location* by computing an expression, storing it in a temporary location, and then retrieving the computed value via indexing. The supported features facilitate the implementation of 83% of the kernels defined in the ONNX specification. However, features like *indirection*, *data-dependent range*, *dependent iteration*, and *general control flow*, while offering further enhancements to IR expressiveness, were deliberately excluded due to the significant challenges they pose in maintaining semantic preservation.

2.2 Transformations

In Figure 5, we demonstrate the effect of transformations on the generated code. Unlike traditional compiler passes, transformations in PerfDojo operate on individual locations within the code. To resolve ambiguity when multiple locations are available (e.g., there can be multiple unrelated scopes that can be fused with `join_scopes`), transformations must be supplied with a unique reference to the specific code location where they should be applied. In the provided example, `join_scopes` identifies the target scope by its depth `0` within the scope nest that encloses the operation with the output array name `t`. Then, it fuses this target scope with the scope that immediately follows it. `reuse_dims` identifies the affected code location based on the buffer name `t` and the dimension at index `0`.

We make the effect of each transformation simple enough that human engineers can verify the correctness of transformations and algorithmically describe the verification process. This enables a new program optimization paradigm that eliminates the need for expert knowledge about architecture to ensure program validity. Consequently, it allows RL to learn about architecture by experimenting with transformations without requiring any prior knowledge.

To obtain the list of program locations where transformations can be applied, we equip each transformation with corresponding applicability detection functionality. In addition to finding appropriate locations for a transformation, this functionality also filters out cases that would cause semantic violations. For instance, the semantic preservation failure demonstrated in Figure 5 is automatically avoided by checking if the affected buffer dimension is used in more than one scope.

Our framework ensures semantic preservation by embedding correctness analyses directly within the logic for identifying transformation locations. This integrated design prevents the application of semantically invalid transformations, removing the burden of correctness verification from the user.

This guarantee is founded upon established compiler principles. The applicability of each transformation is determined by prerequisite analyses, including traditional data dependency analysis, which are encoded into the logic for identifying applicable transformations. We empirically validate the implementation of these applicability rules by numerically comparing the output of each transformed program against its original version.

In Figure 4, we show the path in the program transformation graph that leads to an efficient softmax implementation. Finding such a path is, in essence, the goal of PerfDojo's game. While multiple efficient implementations may exist, reaching them is not trivial. At each implementation variant (node of the graph), there is a different set of applicable transformations, numbering in the hundreds, and only one of them should be selected. As demonstrated in the example, a total of 56 transformations were required to reach this particular implementation.

One of the key features requested for the PerfDojo design is the presence of efficient implementations that are reachable by humans through a manual code optimization process. To verify that our transformation space design includes these implementations, we evaluate the performance of kernel implementations obtained through a manual code optimization process and compare them against state-of-the-art libraries in Figure 10.

Overall, PerfDojo implements a comprehensive suite of common optimizations, including loop and instruction reordering, loop fusion, and vectorization. It also supports loop parallelization for CPU and GPU architectures and enables memory layout transformations such as dimension reordering, padding, and storage type selection. Additionally, it supports experimental hardware features of the Snitch architecture (Section 4.1), such as Stream Semantic Registers (SSR) and Floating-Point Repetition (FREP).

3 PerfLLM: Learning the Performance Game

Motivated by the design and inherent capabilities of our intermediate representation (IR), and inspired by recent works that successfully apply reinforcement learning (RL) to code optimization [10, 52, 60], we recognized that RL offers an effective approach to navigate the vast combinatorial search spaces generated by the number of available transformations. The sheer number of possible transformation sequences at each step presents a significant challenge, one that RL is particularly well-suited to tackle. In this section, we outline the core principles of RL and detail how we formulate the performance game as an RL problem to efficiently discover high-performance optimizations and transformations.

Markov Decision Process. Most reinforcement learning (RL) problems can be modeled as Markov Decision Processes (MDPs), which capture environments where outcomes are determined by both stochastic dynamics and an agent's decisions. An MDP is defined by a *state space* ($\mathcal{S}$), an *action space* ($\mathcal{A}$), a state transition function (p), a *reward function* (r), a discount factor (γ), and a policy (π) [39, 56]. When an agent interacts with an MDP, a sequence

of trajectory can be obtained [39]: $s_0, a_0, r_1, s_1, a_1, r_2, \ldots$, where the subscript t denotes the current time step, $s_t \in \mathcal{S}$, $s_t \in \mathcal{A}$. The *return* after time-step t can be calculated as the sum of cumulative discounted rewards that are received till the termination time T: $R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$. Under the Markov property, the agent's goal becomes maximizing R_t. This objective is formalized through the *Bellman optimality equations*, which recursively relate the optimal *state-value function*,

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}, r} p(s', r|s, a) \left[r + \gamma V^*(s') \right], \tag{1}$$

and the optimal *state-action value function*,

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}, r} p(s', r|s, a) \left[r + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a') \right]. \tag{2}$$

These equations form the basis for iterative methods like value iteration and policy iteration, enabling the derivation of the optimal policy π^* via the equation

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} Q^*(s, a). \tag{3}$$

The ϵ-*greedy policy* is a simple yet effective method for balancing exploration and exploitation in reinforcement learning. Under this approach, the agent selects the best-known action (the "greedy" action) with a high probability—typically $1 - \epsilon$ plus a small share divided among all actions—and chooses a random action with a probability of ϵ. This strategy encourages the agent to explore new actions, especially during early training when ϵ is set high, and gradually shifts towards exploiting known good actions as ϵ decays, thereby facilitating faster convergence to an optimal policy.

3.1 RL Formulation

When formulating real-world problems as reinforcement learning tasks, the design of the state space, action space, and reward function is essential.

In PerfLLM (Figure 1a), the primary role of LLM is to encode the PerfDojo program representation into a numerical embedding vector. This embedding captures the program's state to provide a compact, high-level representation of the corresponding kernel configuration, eliminating the need for manual feature engineering. Mathematically, if the kernel at time step t is denoted by k_t, the corresponding state is given by $s_t = E(k_t)$, where $E(\cdot)$ represents the embedding function.

Designing the action space presents a unique challenge due to the large, and dynamically changing, set of available transformations. To address this, we represent each action as the concatenation of the embedding of a kernel before a transformation and the embedding after the transformation is applied. In this framework, the *stop* action is naturally defined by concatenating two identical embeddings, indicating no change in state.

Constructing a suitable reward function also proved complex. Our initial approach, which was based on speedup relative to the kernel's prior state, led to undesirable behaviors, such as overly conservative transformation choices or exploitation of the reward mechanism through cyclic performance degradation and recovery. To address these issues, we defined the reward as $r = \frac{c}{T_{s_i}}$, where c is a scaling constant and T_{s_i} is the runtime of the kernel after a transformation. This formulation incentivizes actions that improve the kernel's performance without relying on some absolute runtime as a reference. Furthermore, by providing rewards after each transformation rather than only at the end of an entire optimization episode, we mitigate problems caused by sparse rewards.

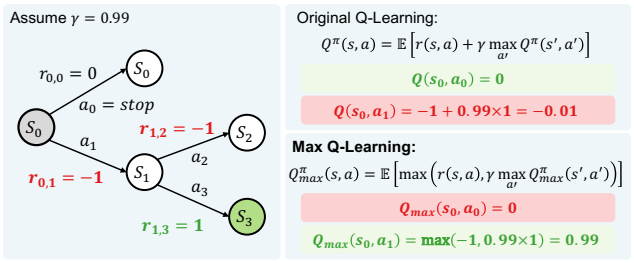


Figure 6: An example comparing the Q-value updates in original Q-Learning and Max Q-Learning. The best achievable state S_3, highlighted in green, demonstrates how Max Q-Learning explicitly prioritizes trajectories leading to higher peak rewards, thereby selecting action a_1, whereas Original Q-Learning selects the immediate "stop" action a_0.

3.2 Deep Q-Learning

Q-learning is a foundational, model-free reinforcement learning algorithm that iteratively updates an action-value function $Q(s, a)$ to learn an optimal policy [55]. It does so by refining its estimates of expected returns based on observed rewards and the maximum future Q-value. In tabular settings, each state-action pair is maintained explicitly, but this becomes infeasible for large or continuous state spaces. Deep Q-learning (DQN) addresses this limitation by using a neural network to approximate the Q-function, enabling an agent to learn effectively even when the state space is too large to enumerate [26].

An alternative class of methods is policy gradient algorithms, which optimize policies by directly estimating the gradient of expected returns, but they often suffer from high variance and sample inefficiency [18, 36, 40]. These issues are particularly acute in environments with large, discrete action spaces. For our task, which involves discrete and computationally expensive transformation evaluations, these drawbacks are especially problematic. Consequently, policy gradient methods are less appropriate for PerfLLM, where stable and sample-efficient learning is essential.

Max Q-Learning. Through experimentation, we found that using the original Q-learning objective was not effective in our scenario because traditional Q-learning is designed to maximize the expected cumulative reward, averaging outcomes across many trajectories. In our context, however, the goal is to identify the single best trajectory that yields the highest possible reward, making standard Q-learning unsuitable. To better address this, we adopted the Max Q-learning approach proposed by Gottipati et al. [17], which directly focuses on maximizing the best achievable reward within an episode. This modification introduces a revised Bellman equation known as the *max-Bellman* equation:

$$Q^\pi_{\max}(s, a) = \mathbb{E}\Big[\max(r(s, a),\ \gamma\, Q^\pi_{\max}(s', a'))\Big]. \tag{4}$$

Figure 6 illustrates a simple example highlighting this difference. Starting from state S_0, the traditional Q-learning objective would select action a_0 (to stop immediately) due to its higher expected cumulative reward. However, Max Q-learning evaluates action a_1 (leading to state S_3) more favorably because it explicitly prioritizes trajectories that achieve higher peak rewards. Consequently, in our PerfLLM implementation, we utilize the max-Bellman equation to train the RL agent, guiding it toward discovering and applying transformations that lead to the most optimal code performance.

3.3 RL Training Techniques

Through extensive hyperparameter tuning and empirical evaluation, we identified and applied the following set of techniques to enhance RL agent training. Other commonly employed methods in reinforcement learning literature, such as prioritized experience replay [32] and noisy networks [13], were excluded, as our experiments indicated that these approaches generally did not provide meaningful performance gains in our specific setting.

Experience Replay. Experience replay is an essential technique for overcoming two challenges in training deep Q-networks [26]. First, it enhances sample efficiency by storing each transition in a replay buffer, allowing the network to update its parameters multiple times using the same experience rather than discarding it after a single use. Secondly, it combats the instability caused by the high correlation of sequentially collected data. By randomly sampling mini-batches from the replay buffer, experience replay breaks the temporal correlations between samples, leading to more stable gradient updates and preventing oscillations or divergence in the network parameters.

Double DQN. Double DQN, introduced by van Hasselt et al. [48], improves upon standard DQN by addressing training instability and overestimation of Q-values. In traditional DQN, the target is computed as

$$Q(a,s) = r(s,a) + \gamma \max_{a'} Q(s', a'; \theta), \tag{5}$$

where θ represents the parameters of the policy network, using the same network for both action selection and evaluation, which can lead to unstable updates. Double DQN decouples these by selecting the best action using the current network and evaluating it with a separate target network, whose parameters are denoted as θ^-, yielding the expression

$$Q(s,a)^{\text{DoubleDQN}} = r(s,a) + \gamma Q(s', \operatorname*{argmax}_a Q(s', a; \theta); \theta^-). \tag{6}$$

This approach stabilizes training and reduces bias, leading to faster, more reliable convergence [27].

Dueling Network. The dueling network architecture enhances deep Q-learning by decomposing the Q-value function into two distinct streams: one that estimates the state value $V(s)$ and another that calculates the advantage $A(s,a)$ for each action [54]. Instead of directly predicting $Q(s,a)$, the network computes these components separately and then combines them to obtain the final Q-values. This separation allows the model to learn which states are valuable without needing to learn the effect of each action in every state, which is particularly useful in environments where many actions yield similar outcomes. By isolating the value of a state from the advantage of individual actions, the dueling network architecture improves sample efficiency, stabilizes learning, and often leads to faster convergence in complex scenarios.

4 Winning the Performance Game

We begin our evaluation by demonstrating that heuristic-based optimization implemented in PerfDojo is time-competitive with existing methods. In Section 4.1, we use simulation to evaluate its support for novel hardware by targeting an experimental RISC-V design with custom extensions. Subsequently, in Section 4.2, we show that our heuristic-driven search on conventional hardware outperforms state-of-the-art approaches.

Following this, Section 4.3 evaluates our primary contribution, PerfLLM (Section 3), which discovers high-performance implementations without relying on hardware-specific heuristics to guide the search. In this approach, hardware knowledge is exposed to the search algorithm only as a library of transformations, without any a priori information on their expected performance impact.

4.1 Heuristic pass for RISC-V

We streamline the process of performance optimization on novel hardware, demonstrating our approach by leveraging the capabilities of the Snitch architecture [58]. Snitch extends [33] the open RISC-V ISA and delivers 2× higher energy efficiency in floating-point computations compared to CPUs [6] and GPUs [11] of the same generation. The energy efficiency of the Snitch [58] architecture is achieved by two RISC-V ISA extensions: stream semantic registers (SSR) [33] and floating-point repetition (FREP).

We conduct our experiments using deterministic cycle-accurate simulation with the Verilator model of Snitch-cluster. We compare the achieved performance against the theoretical compute peak. In most cases, the kernels we analyze involve either floating-point or integer arithmetic exclusively. Therefore, estimating the peak achievable instructions per cycle as 1.0, we calculate the theoretical peak by counting the number of required arithmetic operations.

In Figure 7, we examine three strategies applied during an optimization pass to explore the transformation space. The *naive* strategy imitates the programmer's actions without extensive architectural insight, aiming only to merge scopes and reuse buffers as much as possible. The *greedy* approach incorporates hardware-aware transformations and applies them exhaustively, assuming that their usage is always beneficial. The geometric mean speedup over the *naive* is 46%.

The *heuristic* strategy leverages hardware expertise by analyzing elements within the IR to select transformations that optimize performance. For instance, kernels optimized using the *greedy* strategy typically achieve only about 25% of peak performance due to a 4-cycle instruction pipeline latency. To overcome this bottleneck, the *heuristic* strategy seeks to mitigate latency by tiling the outermost loops in each loop nest to a size of 4, whenever possible. For example, if the initial loop nest has dimensions `[N,D1,D2]` (from outermost to innermost), the transformation reshapes it to `[N/4,4,D1,D2]`. The dimension of size 4 is then repositioned to the innermost part of the loop hierarchy, resulting in a structure of `[N/4,D1,D2,4]`. Finally, unrolling is applied to this innermost

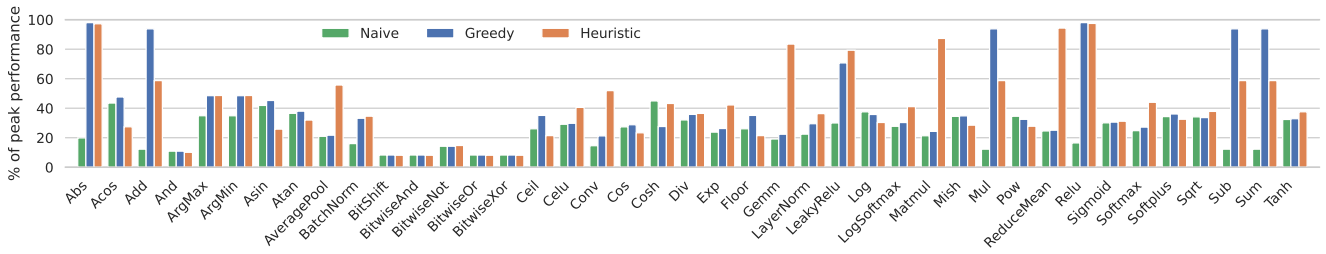


Figure 7: Comparison of micro-kernel performance achieved through passes implementing various transformation strategies for the Snitch RISC-V extensions. *Naive* applies loop fusion and memory reuse until exhaustion. *Greedy* extends the *naive* pass with hardware-specific transformations. *Heuristic* is implemented by a hardware expert as a function that accounts for the structure of the program.

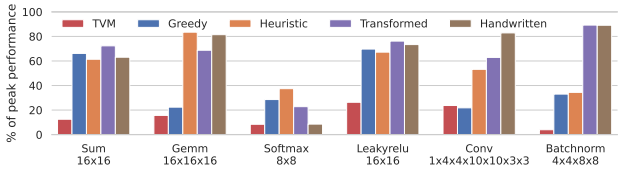


Figure 8: Performance of micro-kernels obtained through automated passes (*greedy*, *heuristic*), manual transformation-centric optimization (*transformed*), TVM 0.11.1, and *hand-written* C and assembly for the Snitch RISC-V extensions.

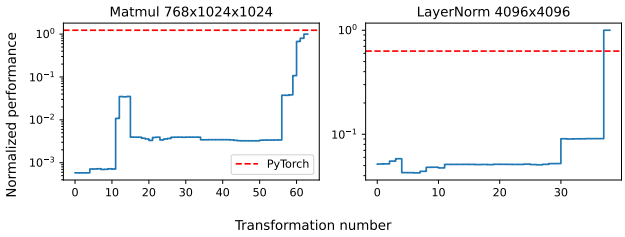


Figure 9: Performance during manual code transformation process.

dimension. This *heuristic* strategy yields a geometric mean speedup of 58% over the *naive*.

In Figure 8, we evaluate the performance of micro-kernel implementations. We compare implementations generated through transformations with those produced using TVM and manually crafted by Snitch cluster developers. Handwritten implementations heavily utilize inline assembly to leverage Snitch features. TVM is provided only as a reference since it does not consider Snitch features, limiting its optimization capabilities. The geometric mean speedup of *transformed* over *handwritten* implementations is 13%.

Our approach is particularly advantageous for quickly examining potential transformations, as memory accesses are automatically handled by the code generator. This task is typically challenging, especially when configuring SSR and FREP extensions. Our streamlined transformation management enables faster identification of loops that can utilize these extensions. Once such loops are identified, users of this transformation-centric pipeline do not need assembly knowledge to enable these features. Finally, our approach supports applying transformations exhaustively, aiding in uncovering optimization opportunities that may not be immediately apparent.

4.2 Heuristic search

Observing the manual transformation process, illustrated by the performance impact of each transformation in Figure 9, we encountered challenges in using traditional search methods such as greedy search and simulated annealing. These challenges include traversing and escaping local minima, as well as navigating large plateaus of equivalent performance due to transformations that do not immediately affect performance but enable critical optimizations later in the process.

4.2.1 Structure of the Search Graph. Initially, we considered a straightforward approach to structuring the search graph by aligning it with the transformation graph. In this setup, the structure of the search graph mirrors the edges of the transformation graph, which we refer to as *edges*-based.

To address the challenges of navigating the search space, we propose an alternative *heuristic*-based method inspired by the expert hand-tuning process. In this approach, the transformation sequence is not constructed sequentially from start to finish. Instead, an initial complete sequence is generated as a candidate and then iteratively refined by modifying selected transformations at arbitrary points, leaving other transformations unaffected. To support this search process, we define a heuristic function specific to the target hardware, which suggests alternative candidate sequences for a given transformation sequence.

4.2.2 Search Method. For the search, we implement two strategies that differ in their transformation selection process and cost definition. The first strategy is a global random *sampling* over all previously encountered programs, with selection probabilities based on the costs of past evaluations. Here, we define the cost of a transformation sequence as the runtime of its parent in the search graph. This approach avoids allocating time budget to evaluate children of weakly performing candidates, in contrast to defining a program's cost as its own runtime. The second strategy relies on simulated

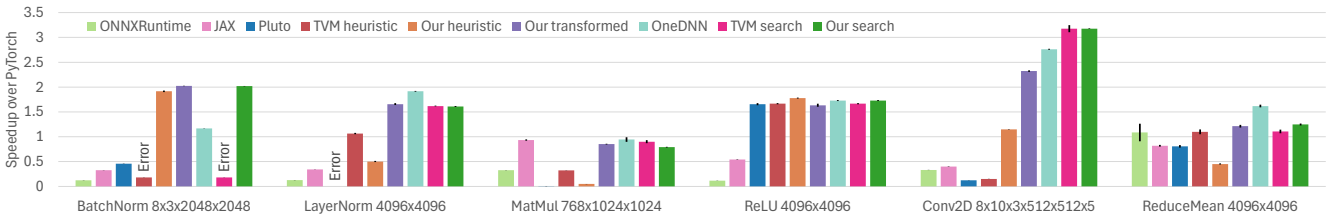


Figure 10: Kernel performance across various frameworks and libraries on x86. The *heuristic* version performs a single program evaluation pass, whereas the *search* version continues until reaching a limit of 1000 evaluations, after which the search terminates. The *transformed* version is derived by applying transformations manually.

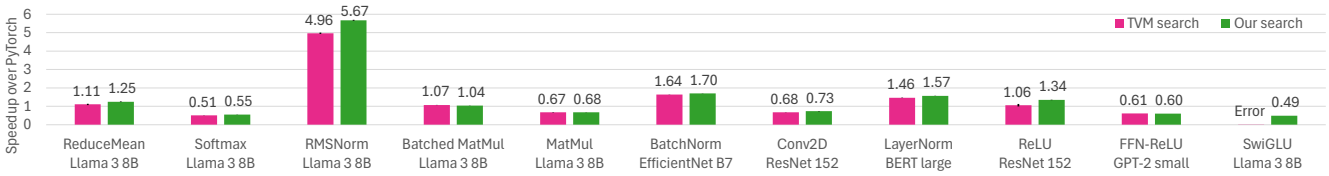


Figure 11: Kernel performance across various shapes in existing models after 1000 auto-tuning evaluations on x86. Excluding SwiGLU, where the TVM auto-scheduler fails to produce a valid schedule, our approach achieves a 7.6% geometric mean speedup over it.

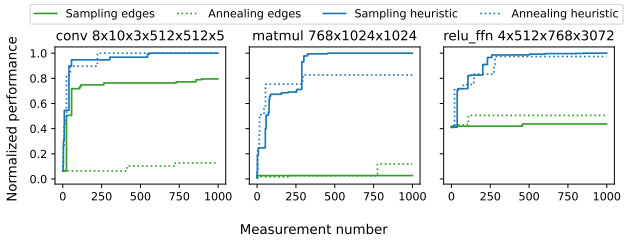


Figure 12: Comparison of convergence speed between simulated annealing and random sampling across differently structured search spaces, based on *edges* within the transformation graph or *heuristic* transitions among transformation candidates.

annealing, which, unlike the first method, allows us to define a program's cost directly as its runtime, thus avoiding the limitations of the sampling-based approach.

In Figure 12, we compare various search methods and search space structures. The use of heuristics emerges as a decisive factor in the speed of performance convergence, highlighting the importance of supplying heuristics incorporating expert knowledge about hardware specifics.

4.2.3 Performance on x86. Evaluation of our heuristic search was performed on all 18 cores of an Intel Xeon CPU E5-2695 v4 with hyper-threading disabled. The following software versions were used: PyTorch 2.3.1, ONNXRuntime 1.18.1, JAX 0.4.31, OneDNN 3.5.3, TVM 0.16.0, Pluto [5] 0.12.0, and Clang/LLVM 18.1.8. Each evaluation incorporated warmup iterations, with each iteration using new input data. All kernels from PyTorch and JAX utilized their respective just-in-time (JIT) capabilities. ONNXRuntime evaluation employed its default execution provider. We use the recommended `--parallel --tile` flags for Pluto with default tile sizes.

Figures 10 and 11 demonstrate that the introduced abstractions do not limit attainable performance and remain competitive with both highly optimized handwritten libraries and state-of-the-art auto-tuning on a well-studied CPU architecture.

When using sizes derived from existing models (Figure 11), the performance gains from auto-tuning are not consistently superior to those of PyTorch, particularly evident in the convolution kernel. However, with less common sizes (Figure 10), auto-tuning can surpass handwritten libraries, suggesting limited optimization for these uncommon sizes. This highlights an advantage of the transformation-centric approach over the library-centric approach, as it provides flexibility for discovering more efficient implementations, even on existing architectures.

The results in Figure 11 demonstrate that our method's performance is frequently comparable to that of TVM. We attribute this to both approaches targeting similar computational and memory bandwidth thresholds while leveraging the same robust LLVM backend, which features well-tuned heuristics for the evaluated architecture. A greater performance disparity is evident on architectures for which the compiler heuristics are less optimized, as exemplified in Section 4.3 (Figure 1b).

We observed several cases where existing frameworks struggled with schedule validation. For kernels such as BatchNorm and SwiGLU, after 1000 auto-tuning iterations, the TVM auto-scheduler produced no valid schedules. Attempts were either rejected for exceeding a 15-second compilation timeout or terminated with a runtime error. More critically, Pluto's optimization of the LayerNorm kernel failed numerical validation. These challenges highlight the need for robust correctness guarantees in optimization frameworks.

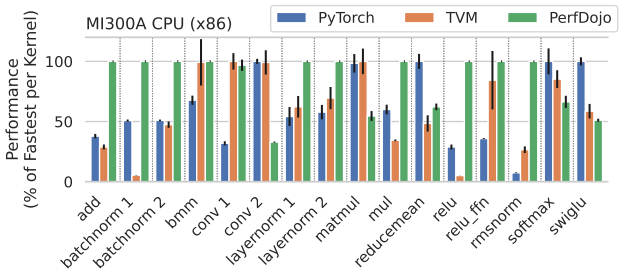


Figure 13: PerfDojo performance compared to PyTorch and TVM. The geometric mean speedup of kernels optimized with PerfDojo on MI300A is 1.56× over PyTorch and 1.80× over TVM.

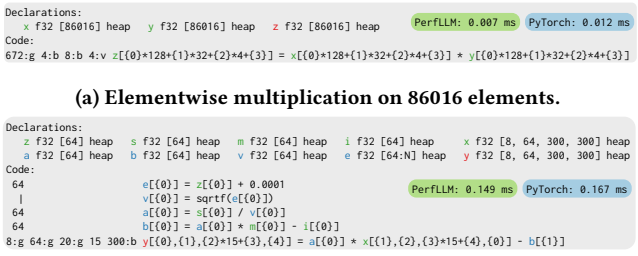
Table 3: ML operators optimized using PerfLLM.

Label	Input shape	Description
add	3072×4096	Elementwise addition
batchnorm 1	8×3×2048×2048	Batch Normalization
batchnorm 2	8×64×300×300	Batch Normalization
bmm	192×256×128×256	Batched Matrix Multiplication
conv 1	8×10×3×512×512×5	2D Convolution
conv 2	8×64×64×56×56×3	2D convolution
layernorm 1	16384×1024	Layer Normalization
layernorm 2	4096×4096	Layer Normalization
matmul	768×1024×1024	Matrix Multiplication
mul	6×14336	Elementwise multiplication
reducemean	4096×4096	Average along axis
relu	4096×4096	Rectified Linear Unit (ReLU)
relu_ffn	8×64×112×112	ReLU+FeedForward Network
rmsnorm	3072×4096	Root Mean Square Normalization
softmax	24576×512	Softmax
swiglu	1×256×4096×448	SwiGLU activation function

4.3 Search with PerfLLM

We evaluate the performance of PerfLLM on two accelerators: the AMD MI300A and the Nvidia GH200. The following software versions were used: GCC 7.5.0, Clang 19.1.3, CUDA 12.6, ROCm 6.2.4, PyTorch 2.6.0, and TVM 0.19.0.

We present the search results of PerfLLM across a set of deep learning kernels (Table 3) in Figures 1b and 13. For a significant portion of the considered kernels, the TVM search was unable to identify any valid schedules; consequently, in these instances, we had to use the default schedule. This was primarily due to exceeding the runtime timeout, which TVM defaults to 10 seconds, a limit we also used in our search. This is not unusual behavior and has been frequently reported by TVM users ([42–47]). Less frequently, schedules were rejected due to compilation timeouts, often caused by large unrolling factors chosen by TVM. Finally, in a few rare instances, kernels generated by TVM failed numerical verification. Since TVM does not perform this verification itself, assuming all generated schedules are correct, we believe these failures are software bugs rather than a fundamental limitation.



(a) Elementwise multiplication on 86016 elements.

(b) Batch normalization with input size $N = 8$ (batch size), $C = 64$ (features), and $H = W = 300$ (height/width).

Figure 14: An overview of the GPU kernel implementations discovered by PerfLLM and their performance.

The search process with PerfLLM is more computationally expensive than heuristic-guided methods (Sections 4.1 and 4.2), resulting in an increase from 5× to 100× in runtime. For perspective, based on an eight-hour optimization time limit per kernel, we extrapolate that tuning a full library of approximately 160 ONNX operators would require an estimated 1280 node-hours. Nevertheless, this one-time investment represents a substantial saving compared to the engineering effort required to manually achieve a comparable level of performance on new hardware.

Next, we describe the specific kernels discovered by RL that outperform their PyTorch implementations on the GPU.

Elementwise multiplication. Despite its trivial implementation, the RL-discovered variant outperformed PyTorch by 1.62× and TVM by 1.22× on MI300A, and on GH200 (Figure 14a), it achieved a 1.71× speedup over PyTorch and 3× over TVM. This demonstrates that RL can identify common optimization techniques used by humans: it vectorized the innermost loop (size 4) to enable 128-bit loads instead of 32-bit, and set the total block size equal to the warp size ($32 = 4 \times 8$), which is a standard practice. We attribute the speedup partly to PyTorch's use of padding for general input sizes, which reduces efficiency in this specific case.

Batch normalization. On GH200, PerfLLM reached 92% of PyTorch's performance, while TVM failed to yield a valid schedule. On MI300A (Figure 14b), however, PerfLLM's implementation surpassed PyTorch by 1.12× and TVM by 1.76×. It achieved this by sequentially computing the temporaries e, v, a, and b on the CPU before launching the CUDA kernel. Since the input's height and width are not multiples of the 64-thread wavefront size, both our method and PyTorch incur redundant computation on padded elements. To mitigate this, PerfLLM selected a block size of 300, padding it to 320 by using 5 wavefronts.

5 Related Work

Halide [30] introduced programmable schedules for image pipelines, a foundational idea that influenced the subsequent development of TVM. To overcome the difficulty of integrating highly hardware-specific sketch derivation rules into Ansor [59] and to achieve performance comparable to handwritten libraries, a later work [8] proposed embedding problem- and hardware-specific code generation within TVM. Nevertheless, this dependence on a code generator

tailored to a narrow problem set limits its applicability to a broader spectrum of user applications.

While TVM has seen significant development, Bolt [57] demonstrated that Ansor's performance on GPUs reached only approximately 10% of the efficiency of vendor-optimized cuBLAS. To address this performance gap, Bolt integrated CUTLASS as a configurable micro-kernel. Building upon these efforts, MetaSchedule [35] was introduced to refine the definition of the schedule space within TVM, aiming to overcome prior limitations. Further extending TVM's framework, TensorIR [12] explores and enumerates mappings of low-level iteration blocks to hardware intrinsics, such as Nvidia's Tensor Cores and specialized ARM vector instructions (e.g., sdot). However, this approach demonstrated performance improvements limited to convolution and GEMM kernels.

Triton [41] specifically targeted programming neural networks on NVIDIA GPUs. It automated parts of scheduling, such as memory coalescing and shared memory management, while leaving users to expose autotunable parameters of kernel implementations. Triton serves as one of the backends for TorchInductor to compile PyTorch [29] models.

Exocompilation [20, 21] demonstrated the Exo language and transformations, along with the effect analysis framework to detect their applicability, aiming to optimize hardware accelerators in close-to-peak utilization regimes. Due to the high expressiveness of the language, effect analysis is not trivial and may end up being too conservative for optimizing a broader set of kernels than the evaluated GEMM and convolution.

ISA Mapper [38] explored mapping linear algebra computations to hardware instructions. However, its scheduling space was closer to the macro-kernel level, involving the management of parallelism and communication.

Polyhedral Compilation. The introduction of the Tensor Comprehensions [49] IR extended the capabilities of the Halide pipeline to serve as a frontend for deep learning tasks, leveraging the polyhedral model to guide optimizations [50].

Tiramisu [2] is a polyhedral compiler integrated with a scheduling language, leveraging the LLVM infrastructure.

MLIR [24] addressed the need to express features of various architectures through domain- and hardware-specific dialects. MLIR was explicitly designed to support polyhedral optimizations. Deep learning frameworks developed by Google, such as TensorFlow [1] and JAX [15], relied on XLA [37] optimizations implemented using the MLIR framework. MLIR [24] provides users with flexibility in customizing optimization passes. Nonetheless, MLIR does not provide an answer to optimization automation.

Overall, a polyhedral IR does not natively support hardware-specific features, making them also unavailable to polyhedral schedulers [28]. While a polyhedral IR can be extended, for instance, with annotations, such extensions are susceptible to the issues of general-purpose IRs. In particular, the expressiveness of such an IR can make it difficult to either prove semantic preservation or ensure that the range of code implementation variants includes efficient programs that utilize hardware-specific knowledge, especially on novel hardware. In contrast, we start defining an IR with semantic preservation in mind and manage to effectively enable hardware-specific features for a significant portion of the deep learning kernels.

6 Discussion and Future Work

Our implementation efforts focused on enabling features that have efficient support from hardware vendors (Table 2). Extending these features is an intriguing direction for future research. Developing a formal theory or classification of these features would help distinguish general control flow from other representational features, providing a clearer foundation for transformation analysis in future frameworks.

7 Conclusion

We address the challenge of implementing and optimizing deep learning kernels on modern hardware architectures. Our study focuses on streamlining and automating this process by leveraging program transformations. We propose a transformation-centric workflow that splits the optimization process into two parts: first, defining transformations and detecting their valid application (without altering program semantics); and second, finding effective sequences of transformations for a particular kernel.

This separation enables the application of machine learning methods for identifying transformation sequences, while also allowing engineers to implement custom heuristics as they become familiar with the architecture and develop the ability to make informed estimations for parameters such as tile sizes. However, with the introduction of PerfLLM, we demonstrate that PerfDojo also supports a fully automated, RL–based approach to constructing transformation sequences.

Leveraging the PerfDojo representation, a heuristic pass achieved a 13% geometric mean gain over hand-optimized RISC-V kernels. On x86, our approach delivered a 7.6% speedup against a state-of-the-art auto-tuner using only greedy search and simulated annealing. For arm, PerfLLM automated optimization to realize a 6.65× speedup over PyTorch without hardware-specific heuristics. These results demonstrate that semantic-preserving transformations facilitate the automated exploration of transformation sequences, paving the way for developing hardware-specific libraries.

Acknowledgments

This work has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 101034126 (EU-Pilot) and the ERC project PSAP under grant agreement No. 101002047. We thank the CSCS supercomputing center for access to compute resources.

The authors utilized AI assistant tools, including ChatGPT and Gemini, for light editing and proofreading throughout this manuscript. All content and ideas remain the original work of the authors.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[3] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[4] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[5] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. 2008. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer.

[6] Matheus Cavalcante, Fabian Schuiki, Florian Zaruba, Michael Schaffner, and Luca Benini. 2019. Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28, 2 (2019), 530–543.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11, 20 (2018).

[8] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring your own codegen to deep learning compiler. *arXiv preprint arXiv:2105.03215* (2021).

[9] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[10] 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. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. arXiv:2109.08267 [cs.PL] https://arxiv.org/abs/2109.08267

[11] Michael Ditty, Ashish Karandikar, and David Reed. 2018. Nvidia's xavier soc. In *Hot chips: a symposium on high performance chips*.

[12] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, et al. 2023. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 804–817.

[13] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. 2019. Noisy Networks for Exploration. arXiv:1706.10295 [cs.LG] https://arxiv.org/abs/1706.10295

[14] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. 2025. Marlin: Mixed-precision auto-regressive parallel inference on large language models. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 239–251.

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

[16] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. 2019. Xilinx adaptive compute acceleration platform: Versaltm architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 84–93.

[17] Sai Krishna Gottipati, Yashaswi Pathak, Rohan Nuttall, Sahir, Raviteja Chunduru, Ahmed Touati, Sriram Ganapathi Subramanian, Matthew E. Taylor, and Sarath Chandar. 2023. Maximum Reward Formulation In Reinforcement Learning. arXiv:2010.03744 [cs.LG] https://arxiv.org/abs/2010.03744

[18] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews)* 42, 6 (2012), 1291–1307.

[19] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[20] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 703–718.

[21] Yuka Ikarashi, Kevin Qian, Samir Droubi, Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. 2025. Exo 2: Growing a Scheduling Language. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 426–444.

[22] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. 2023. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*. 1–14.

[23] Stavros Kalapothas, Manolis Galetakis, Georgios Flamis, Fotis Plessas, and Paris Kitsos. 2023. A survey on risc-v-based machine learning ecosystem. *Information* 14, 2 (2023), 64.

[24] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[25] David Luebke. 2008. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 836–838.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602 [cs.LG] https://arxiv.org/abs/1312.5602

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.

[28] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. Polymage: Automatic optimization for image processing pipelines. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 429–443.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[31] Philipp Schaad, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Alexandros Nikolaos Ziogas, and Torsten Hoefler. 2023. FuzzyFlow: Leveraging Dataflow To Find and Squash Program Optimization Bugs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[32] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. arXiv:1511.05952 [cs.LG] https://arxiv.org/abs/1511.05952

[33] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2020. Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores. *IEEE Trans. Comput.* 70, 2 (2020), 212–227.

[34] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *Advances in Neural Information Processing Systems* 37 (2024), 68658–68685.

[35] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. *Advances in Neural Information Processing Systems* 35 (2022), 35783–35796.

[36] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic policy gradient algorithms. In *International conference on machine learning*. Pmlr, 387–395.

[37] Daniel Snider and Ruofan Liang. 2023. Operator Fusion in XLA: Analysis and Evaluation. *arXiv preprint arXiv:2301.13062* (2023).

[38] Matthew Sotoudeh, Anand Venkat, Michael Anderson, Evangelos Georganas, Alexander Heinecke, and Jason Knight. 2019. ISA mapper: a compute and hardware agnostic deep learning compiler. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 164–173.

[39] R.S. Sutton and A.G. Barto. 2018. *Reinforcement Learning, second edition: An Introduction*. MIT Press. https://books.google.com/books?id=uWV0DwAAQBAJ

[40] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller (Eds.), Vol. 12. MIT Press. https://proceedings.neurips.cc/paper_files/

paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf
[41] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 10–19.
[42] TVM Community Discussion 11001. [n. d.]. Autoscheduler failed to find a valid schedule. https://discuss.tvm.apache.org/t/autoscheduler-failed-to-find-a-valid-schedule/11001
[43] TVM Community Discussion 12268. [n. d.]. Auto-Scheduler Time Out: No Valid Schedule for GEMM. https://discuss.tvm.apache.org/t/auto-scheduler-time-out-no-valid-schedule-for-gemm/12268
[44] TVM Community Discussion 12750. [n. d.]. Auto-Scheduler Cannot Find Any Valid Schedule. https://discuss.tvm.apache.org/t/auto-scheduler-cannot-find-any-valid-schedule/12750
[45] TVM Community Discussion 15238. [n. d.]. Bug: Auto Scheduler cannot find any valid schedule. https://discuss.tvm.apache.org/t/bug-auto-scheduler-cannot-find-any-valid-schedule/15238
[46] TVM GitHub Issue 15206. [n. d.]. Auto Scheduler cannot find any valid schedule. https://github.com/apache/tvm/issues/15206
[47] TVM GitHub Issue 16670. [n. d.]. Auto scheduler Cannot find any valid schedule. https://github.com/apache/tvm/issues/16670
[48] Hado van Hasselt, Arthur Guez, and David Silver. 2015. Deep Reinforcement Learning with Double Q-learning. doi:10.48550/ARXIV.1509.06461
[49] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–26.
[50] Sven Verdoolaege. 2010. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*. Springer, 299–302.
[51] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. 2022. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852* (2022).
[52] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 129–143.
[53] KC Wang. 2023. ARMv8 Architecture and Programming. In *Embedded and Real-Time Operating Systems*. Springer, 505–792.
[54] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. Dueling Network Architectures for Deep Reinforcement Learning. arXiv:1511.06581 [cs.LG] https://arxiv.org/abs/1511.06581
[55] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.
[56] F. Woergoetter and B. Porr. 2008. Reinforcement learning. *Scholarpedia* 3, 3 (2008), 1448. doi:10.4249/scholarpedia.1448 revision #127590.
[57] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems* 4 (2022), 204–216.
[58] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2020. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Trans. Comput.* 70, 11 (2020), 1845–1860.
[59] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.
[60] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 859–873. doi:10.1145/3373376.3378508