A code superoptimizer through neural Monte-Carlo tree search

Wenda Zhou
New York University and Flatiron Institute
wzhou@flatironinstitute.org

Olga Solodova
Princeton University

Ryan P. Adams
Princeton University

Abstract

There are many ways to turn a high-level program into a sequence of instructions consistent with that computation. Selecting the most performant such instruction sequence for a given piece of hardware — optimized compilation — is a central challenge of computer science. Optimizing compilers perform this task through a series of reductions and local transformations (e.g. register allocation, instruction scheduling, peephole optimization) driven by heuristics. A natural and well-explored avenue of research is to replace current hand-written heuristics by data-driven, automatically-designed heuristics which may be obtained from machine learning. We propose a radically different approach, in which we view compilation as a combinatorial optimization problem which consists of finding the optimal (e.g. fastest executing or shortest) sequence of instructions subject to the constraint that it has the semantics of the specified program. We show how this problem can be practically framed as a finite Markov decision process, unlocking a rich space of potential algorithms from reinforcement learning. We implement one such algorithm in particular, an AlphaGo-like distributed neural Monte-Carlo tree search procedure, and demonstrate that it is able to directly generate optimized assembly. Unlike a traditional optimizing compiler, this approach does not rely on an existing library of optimizations to transform the code, but rather directly attempts to generate the most optimal program instruction-by-instruction, taking into account effects including register allocation, instruction scheduling and operation fusion.

1 Introduction

Compiling a high-level program specification into performant machine code is one of the core challenges of computer science, both from a theoretical and practical perspective. It is a difficult task due to the complexity of modern hardware architectures; different sequences of machine instruction—even if they generate identical outputs—can have wildly different costs in terms of, e.g., execution time, code size, and energy consumption. Moreover, among the space of possible instruction sequences only a tiny corner will meet the high-level specification. Paradoxically, the space of instruction sequences is so combinatorially enormous that even this tiny corner of specification-satisfying sequences is still often so large as to make the code optimization problem intractable.

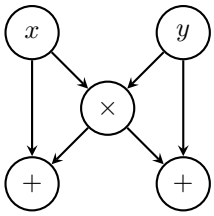
Nevertheless, optimized machine code is a prize worth seeking, as the economic and environmental cost of software—particularly machine learning workloads—increases. Even small performance improvements in extremely "hot" code paths such as matrix multiplication could have a significant impact on data center power consumption and the battery life of edge-computing devices.

In this work, we tackle the optimal code generation problem via neural network-accelerated Monte Carlo tree search, in a fashion similar to that used by AlphaGo [26]. Key to this approach is our novel framing of the code generation task as a Markov decision process (MDP). Our MDP approach is unique in that it guarantees that valid (specification-satisfying) programs will be generated, enabling

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

```
def fn(x, y):
  z = x * y
  return x + z, y + z
```

(a) Initial function definition



(b) Graph representation

```
load %1, [x]
load %2, [y]
mul %3, %1, %2
add %4, %1, %3
add %5, %2, %3
store [o1], %4
store [o2], %5
```

(c) Possible assembly sequence

Figure 1: The optimized compilation problem for basic blocks

all of the effort of the search to be concentrated on the problem of optimizing performance. As mentioned above, the complexity of the interaction between hardware and software results in effects that can have long-range temporal dependence (e.g., pipelined architectures, register allocation, caching); the problem of learning and planning in an MDP—performing temporal credit assignment of costs/rewards—aligns perfectly with this challenge, strongly motivating our approach. Indeed, games such as Go have precisely this character: moves made early in the game can have long-range effects that must be reasoned about. We seek to turn the code generation task into a game that is possible for a learning system to play with superhuman performance.

In recent years, various machine learning and reinforcement learning methods have been applied to the problem of optimized compilation with numerous strategies. Our approach is most similar to superoptimization Massalin [16], and directly attempts to generate hardware instructions, optimizing aspects such as instruction selection, register allocation and instruction scheduling in an end-to-end fashion. We refer the reader to appendix A for a more comprehensive overview of related work.

2 Formal problem description

Given a program specification S, the task of optimized compilation may be generically described as the generation of a program P with minimum cost $C(P)$ which satisfies the specification S. One may directly attempt to formalize the above problem as a combinatorial optimization problem:

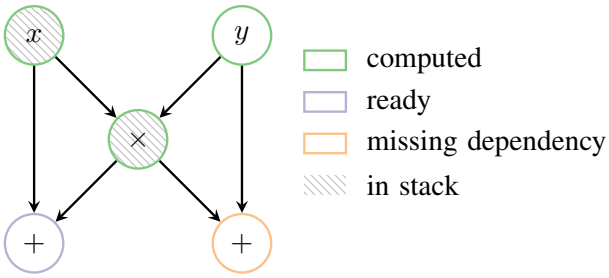
$$\min_P C(P) \quad \text{s.t. } P \text{ satisfies } S. \tag{1}$$

However, this formulation is impractical, as the space of all programs is large and an extremely small portion of programs will satisfy the specification. In general, even producing a single such program will require specific algorithms which examine S in detail. In the context of this paper, we assume that the specification S is given as a circuit, that is, a directed acyclic graph of operations. The program P is represented as a sequence of assembly (machine) instructions, and C may be some estimated or measured performance, or the size of the program. Figure 1 illustrates the problem.

We turn to reinforcement learning in an attempt to learn strategies to construct the problem in a piecewise fashion. By defining an appropriate environment (formalized as a *Markov Decision Process*, or MDP), we establish a correspondence between a sequence of actions taken in the environment and a sequence of instructions computing the specified program. We refer the reader to appendix B for a formal description of MDPs. Our proposed MDP possesses the following desirable properties: 1) it is finite, 2) it has a single absorbing state, and 3) all sequences of actions lead to that absorbing state.

3 The code generation MDP

We observe that the (logical) execution of a program in a processor may be interpreted as a trajectory in a specific MDP by defining the following : 1) The state $s \in \mathcal{S}$ corresponds to the logical state of the processor after a given sequence of instructions (e.g., the contents of the memory and of the registers). 2) The action $a \in \mathcal{A}$ corresponds to the instruction being executed. 3) The transition function $\mathcal{T}$ is given by the effect on the state of the processor when executing the instruction. In principle, we could attempt to directly learn using this code execution MDP by defining an appropriate reward function $\mathcal{R}$, such as the execution time of an instruction from a given state. If the action space is not constrained, this suffers from many of the same problems as a direct attempt to solve eq. (1), for example, a vanishingly small proportion of instructions satisfy the specification.



(a) Example representation of code generation MDP state.

Node	Load	Store	Execute
x	✗	✓	✗
y	✓	✗	✗
xy	✗	✓	✗
$xy + x$	✗	✗	✓
$xy + y$	✗	✗	✗

(b) Available actions for MDP state.

Figure 2: State and actions of the code generation MDP

Our *code generation MDP* is inspired by the observation above, but attempts to constrain the possible actions in order to ensure that 1) the sequence of actions taken leads to a valid program, and 2) the sequence of actions eventually terminates. We enforce (1) by using the structure of our specification S. As we have chosen S to be represented by a circuit, we restrict the instructions at each point in time to those which compute a gate in the circuit. On the other hand, ensuring that all sequences of actions eventually terminate in a valid program is more difficult. One difficult case appears in the state where all registers are exhausted, but no valid gate may be computed from the inputs currently in the registers. To make progress, registers must be spilled to load relevant inputs to the remaining gates; at the same time, we must restrict this in order to prevent infinite loops whereby a value is repeatedly loaded and spilled. Such a condition is difficult to express and enforce. We proceed by defining our MDP through a processor with an infinite stack, rather than a register-based processor. By using an infinite-stack model, we bypass the need to explicitly model spills, and are able to define an MDP with the desired properties. Spills are still important, but they can now appear implicitly in the cost model. To connect our abstract stack-based processor with a hardware model, we will then need some additional processing on the back-end, which we discuss in sections 3.1 and 3.2.

Informal description of the code generation MDP Informally, our code generation MDP represents the execution of a stack machine. The actions correspond to pushing values to and popping values from between the stack, and executing arithmetic operations using values on the stack. Operations are restricted based on the set of computations still required to execute the specified circuit S. Figure 2 illustrates the proposed MDP. For a full description of the MDP, see appendix C.

3.1 From MDP actions to assembly instructions

As defined, a sequence of actions in our MDP translates to a sequence of operations in an abstract memory machine. To produce valid assembly code from such a sequence, it is necessary to translate abstract actions which load, store or execute a gate of the circuit into instructions supported by the target hardware. On targets of interest in the present work, there are two steps in this process: 1) selecting the correct instruction to perform the operation specified by the gate, and 2) assigning values to registers. To select the instruction, we design the circuit such that each gate is computable by an instruction on the target hardware. In some cases, there may be more than a single instruction which maps to a given gate (or set thereof). For example, many floating point units contain a fused multiply-add unit (FMA), which computes $a \times b + c$ in a single instruction. We encode such alternatives by making use of temporal options [28]. We refer the reader to appendix D for details.

We perform register assignment using a greedy least-recently-used strategy (see appendix D). Note that this register allocation strategy is not optimal, nor is it intended to be. By carefully selecting a sequence of actions in the MDP, it is possible to implicitly control register allocation, and we intend this optimization to be carried out by the optimization in the MDP.

3.2 Cost function and MDP rewards

To fully define the code generation MDP, it remains to specify the rewards. By construction, every sequence of actions in the MDP leading to the final absorbing state corresponds to a valid sequence of instructions to carry out the computation, hence we need not assign rewards for correctness,

but only for cost. Additionally, we have significant freedom in designing the rewards of the MDP, as it is sufficient to ensure that the total reward received over a sequence of actions $(a_1, \ldots, a_T)$ terminating at the final state corresponds to the (negative) total cost of the associated program P: $\sum_{i=1}^{T} \mathcal{R}(s_i, a_i) = -C(P)$. One possibility is to only emit a single reward corresponding to the negative cost at the end of the episode. However, such sparse reward structure is notoriously difficult to optimize. Instead, we design the reward to reflect the incremental cost of the additional instruction. Although this is highly non-linear due to the out-of-order nature of modern processors, this remains a better heuristic than a single final sparse reward. We refer the reader to appendix E for details.

4 Implementation

To solve the code generation MDP, we use a neural-guided Monte-Carlo tree search. We implement a distributed neural guided MCTS system similar to that of Silver et al. [26]. We refer the reader to appendix G for an overview of neural-guided MCTS and a description of the main aspects of our implementation.

5 Results

We use our method to optimize a sample of numerically intensive inner loops. Such subroutines may often be found as the innermost loop of numerically intensive code on large desktop and server systems, or may also be used on the "edge" in devices such as micro-controllers in control systems. The smallest circuit contains 35 gates, and the largest 64 gates. For a description of chosen problems, see appendix I. Our model makes no assumption towards the target architecture, except through the cost function which informs the model of the runtime of a program. We thus use of the same model for optimizing the program for a low-power embedded ARM Cortex-M4 CPU, and a server Intel Skylake-family processor. To estimate the cost of programs on ARM, we use a hand-written cost estimate based on the characteristics of the Cortex-M4 FPU described in ARM [3]. To estimate performance on the server Intel processor, we use the open-source tool OSACA [14].

We compare against `gcc` for each target architecture, tuned for maximal performance and given the same fast-math semantics (which in particular, allows reordering of sums and fused multiply-add). We also compare against Souper Sasnauskas et al. [23] for the x86 target, although in this case this superoptimizer is unable to find a missed optimization due to the relative simple nature of the programs. We outperform GCC when targeting Arm (both on estimated and measured performance). When targeting x86, however, we observe a discrepancy between the estimated and measured performance. Although we are on par (or better) in terms of estimated performance, GCC obtains superior measured performance, suggesting some type of sim2real gap not capture by our performance model. See appendix J for further discussion, and appendix K for full details.

One consequence of our assumption-free design is that we may optimize towards different cost functions which need not measure performance. We experiment with the problem of optimizing for code size on x86, which combines the need for the program to have few instructions and the need to choose instructions and registers in a fashion so as reduce the encoded instruction size. Table 3 demonstrates that we are able to achieve results competitive with gcc when the latter is tuned for size.

6 Discussion

We present a new paradigm for applying reinforcement learning to the problem of optimized compilation. We directly generate the optimized program instruction-by-instruction, reducing the need for a large library of optimizations, and we are able to directly gather feedback from an appropriate model for the performance of the target. By making use of the underlying structure of the program specification as a circuit, our method scales to medium-sized basic blocks of 50 to 100 instructions, which are difficult to reach with stochastic search techniques.

We believe that our approach for producing optimized inner blocks, when combined with a higher level program planner such as Halide [2], is ideal to generate high-performance programs on a wide variety of hardware while reducing the human engineering effort required. The last remaining step on such a program, automatic vectorization, however remains one of the hardest. We hope the present paradigm may be extended to cover such cases in future work.

Problem	Ours Est.	Ours Meas.	gcc-arm 11.2 Est.	gcc-arm 11.2 Meas.
Cholesky 4×4	167	175.02(07)	182	182.44(01)
GEMM $4 \times 4 \times 4$ sparse	128	128.15(06)	155	151.14(03)
Gauss-Seidel 5×5 sparse	136	145.13(00)	142	152.98(06)
Neo-Hookean energy 3D	102	115.54(05)	137	140.98(04)
Runge Kutta 4	62	74.70(01)	67	75.98(01)

Table 1: Performance (cycles per iteration) of generated kernels for various problems on an ARM Cortex-M4 processor equipped with a FPU. Measurements taken on an Arduino 33 Nano BLE.

Problem	Ours Est.	Ours Meas.	Ours (tuned) Est.	Ours (tuned) Meas.	gcc 12.1 Est.	gcc 12.1 Meas.	Souper Est.	Souper Meas.
GEMM $4 \times 4 \times 4$ sparse	16.0	17.38(10)	16.0	17.73(10)	19.5	18.86(20)	22.0	21.72(20)
Cholesky 4×4	11.5	37.09(17)	11.5	26.34(11)	12.0	28.67(13)	13.0	29.95(11)
Gauss-Seidel 5×5 sparse	10.0	17.43(09)	10.0	15.60(20)	10.0	14.44(10)	10.0	15.53(10)
Neo-Hookean energy 3D	17.5	21.49(13)	17.5	21.42(21)	19.0	19.68(11)	17.5	19.24(10)
Runge Kutta 4	10.5	18.10(20)	10.5	13.31(08)	12.0	13.92(20)	10.5	13.87(20)

Table 2: Performance (cycles per iteration) of generated kernels for various problems on a Intel Xeon Gold 6234 processor. The "tuned" cost is described in appendix H.5.

Acknowledgments and Disclosure of Funding

This work was partially supported by NSF IIS-2007278. We would like to thank NVIDIA corporation for providing a Nvidia A100 card.

Problem	Ours	gcc 12.1
GEMM $4 \times 4 \times 4$ sparse	359	365
Cholesky 4×4	185	189
Gauss-Seidel 5×5 sparse	135	129
Neo-Hookean energy 3D	280	253
Runge Kutta 4	125	142

Table 3: Program size (bytes) of generated kernels for various problems on a x86 platform.

References

[1] A. Abel and J. Reineke. Accurate throughput prediction of basic blocks on recent Intel microarchitectures. *arXiv e-print*, 2021.

[2] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), jul 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967.

[3] *Cortex-M4 Technical Reference Manual Revision r0p0*. ARM, 2010.

[4] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5), sep 2018. ISSN 0360-0300. doi: 10.1145/3197978.

[5] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021. ISSN 0377-2217.

[6] D. P. Bertsekas and J. N. Tsitsiklis. An analysis of stochastic shortest path problems. *Math. Oper. Res.*, 16(3):580–595, 1991. doi: 10.1287/moor.16.3.580.

[7] R. Bunel, A. Desmaison, M. P. Kumar, P. H. S. Torr, and P. Kohli. Learning to superoptimize programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[8] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. *arXiv e-print*, 2021.

[9] D. Das, S. A. Ahmad, and K. Venkataramanan. Deep learning-based hybrid graph-coloring algorithm for register allocation. *arXiv e-print*, abs/1912.03700, 2019.

[10] C. W. Fraser. A compact, machine-independent peephole optimizer. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 1–6, New York, NY, USA, 1979. Association for Computing Machinery. ISBN 9781450373579. doi: 10.1145/567752.567753.

[11] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73. ACM, 2011. doi: 10.1145/1993498.1993506.

[12] A. Haj-Ali, H. Genc, Q. Huang, W. S. Moses, J. Wawrzynek, K. Asanovic, and I. Stoica. Protuner: Tuning programs with monte carlo tree search. *arXiv e-print*, abs/2005.13685, 2020.

[13] M. Kim, J.-K. Park, and S.-M. Moon. Solving pbqp-based register allocation using deep reinforcement learning. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, 2022. doi: 10.1109/CGO53902.2022.9741272.

[14] J. Laukemann, J. Hammer, J. Hofmann, G. Hager, and G. Wellein. Automated instruction stream throughput prediction for intel and amd microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131, 2018.

[15] A. Liu, J. Chen, M. Yu, Y. Zhai, X. Zhou, and J. Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[16] H. Massalin. Superoptimizer: A look at the smallest program. *SIGARCH Comput. Archit. News*, 15(5):122–126, oct 1987. ISSN 0163-5964. doi: 10.1145/36177.36194.

[17] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Comput. Oper. Res.*, 134:105400, 2021. doi: 10.1016/j.cor.2021.105400.

[18] A. McGovern and J. E. B. Moss. Scheduling straight-line code using reinforcement learning and rollouts. In *Advances in Neural Information Processing Systems 11, [NIPS Conference, Denver, Colorado, USA, November 30 - December 5, 1998]*, pages 903–909. The MIT Press, 1998.

[19] A. McGovern, J. E. B. Moss, and A. G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Mach. Learn.*, 49(2-3):141–160, 2002. doi: 10.1023/A:1017976211990.

[20] T. M. Moerland, J. Broekens, and C. M. Jonker. Model-based reinforcement learning: A survey. *arXiv e-print*, abs/2006.16712, 2020.

[21] J. E. B. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. E. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *Advances in Neural Information Processing Systems 10, [NIPS Conference, Denver, Colorado, USA, 1997]*, pages 929–935. The MIT Press, 1997.

[22] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. *SIGARCH Comput. Archit. News*, 44(2):297–310, mar 2016. ISSN 0163-5964. doi: 10.1145/2980024.2872387.

[23] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr. Souper: A synthesizing superoptimizer. *arXiv e-print*, abs/1711.04422, 2017.

[24] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGARCH Comput. Archit. News*, 41(1):305–316, mar 2013. ISSN 0163-5964. doi: 10.1145/2490301.2451150.

[25] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, Dec 2020. ISSN 1476-4687. doi: 10.1038/s41586-020-03051-4.

[26] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

[27] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

[28] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(99)00052-1.

[29] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li. MLGO: a machine learning guided compiler optimizations framework. *arXiv e-print*, abs/2101.04808, 2021.

[30] S. VenkataKeerthy, S. Jain, R. Aggarwal, A. Cohen, and R. Upadrasta. Rl4real: Reinforcement learning for register allocation. *arXiv e-print*, abs/2204.02013, 2022.

[31] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng. *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System*, page 859–873. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371025.

[32] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. C. Ma, Q. Xu, H. Liu, M. P. Phothilimtha, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon. Transferable graph optimizers for ML compilers. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See sections 5 and 6
   (c) Did you discuss any potential negative societal impacts of your work? [No] The societal impacts of our work are limited due to its nature.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] We have included all obtained experimental artifacts in order to reproduce tables. However, we have no included the training code due to its overly technical and system-specific nature. In lieu, we provide detailed implementation information in appendix H.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See appendices H and K
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See appendix K
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Related Work

We review related work in leveraging machine learning techniques for optimizing compilers.

Reinforcement learning for middle-end compiler optimizations Reinforcement learning has been widely applied as a means of selecting compiler optimizations in the middle-end. Such methods generally integrate with an existing compiler middle-end, and seek to improve or replace heuristics in optimization selection. We refer the interested reader to Ashouri et al. [4] for a survey on machine learning techniques for tuning existing compilers, and to Trofin et al. [29], Cummins et al. [8] for examples of problem formulations when integrating with current compilers. Our approach, by contrast, subsumes the traditional conception of a compiler middle-end and back-end, and attempts to perform both steps together. This enables the joint optimization of back-end concerns (e.g., register allocation, scheduling) and middle-end concerns (instruction selection and transformation).

Learning in the compiler backend Machine learning (and reinforcement learning) has also seen applications to traditional backend tasks of the compiler, such as scheduling [21, 18, 19] or register allocation [9, 30, 13]. Although these ideas share many common aspects, we do not attempt to explicitly model such problems in our approach, but rather rely on our model to automatically learn these tasks, enabling the model to execute trade-offs between all aspects of the program.

Superoptimization A large portion of optimizations in a traditional compiler are performed by matching a fixed set of rules designed to locally transform the instruction sequence (often called peephole optimizations). An ambitious alternate strategy for implementing such optimizations would be to explore the space of instruction sequences through some search procedure, and select the best sequence (according to some specified cost function) among those which have the desired semantics. This approach is often referred to as *superoptimization* as coined by Massalin [16], although such ideas have appeared prior [10]. More recently, stochastic search ideas have been demonstrated by Schkufza et al. [24] with further extensions including pruning [22] and learning-guided [7] search. Synthesis techniques have also been explored, including large-scale implementations integrated with major compilers such as LLVM [11, 23]. Our work shares a conceptual similarity with such synthesizing superoptimizers, though it trades off the ability to discover new equivalent instruction sequences (using e.g., an SMT solver) in exchange for planning and reasoning on much larger scales.

Reinforcement learning for high-level loop planning and scheduling In addition to applications of reinforcement learning in traditional optimizing compilers, there has also been much interest in making use of reinforcement learning to tackle planning and scheduling of high-level operations in an optimal manner. This approach has been particularly fruitful in the context of tensor programs [2, 32, 31, 12], often with applications in the context of neural network inference. We view the current work as complementary to such efforts: these tools are desirable for enabling optimization of the memory and cache hierarchy. On the other hand, they eventually rely on a core micro-kernel which is performance-critical, and for which we present an optimization paradigm.

Neural-guided Monte-Carlo Tree Search (AlphaGo) The reinforcement problem formulated in this paper is by its nature a well-specified MDP which can be accessed with no computational difficulty. This property makes model-based reinforcement learning techniques (see [20] for a general survey), which leverage deep knowledge of the MDP, particularly attractive. Among such, we turn to deep-learning augmented planning techniques. These have demonstrated to incredible success in the context of board games [26, 27] and extended to more general environments Schrittwieser et al. [25].

B Reinforcement learning, MDPs and combinatorial optimization

Reinforcement learning is the problem of learning an optimal decision making policy which maximizes an agent's reward in an environment. Combinatorial optimization can be viewed as falling under this class of problems, provided a suitable environment is constructed from a given combinatorial optimization problem. The goal of an agent in such an environment is then to discover a solution by executing actions. This general methodology for combinatorial optimization has seen many implementations in recent years; we refer the reader to Mazyavkina et al. [17], Bengio et al. [5] for a survey of applications of reinforcement learning to combinatorial optimization. In this paper,

we construct an environment in order to produce a valid program P given a specification S, which we call the *code generation MDP*.

Markov decision process We formalize the environment through a Markov decision process (MDP). Formally, an MDP may be described as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T})$, where $\mathcal{S}$ is a set denoting the state space, $\mathcal{A}$ a set denoting the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function[1], and $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function. The goal of reinforcement learning is to obtain a policy $\pi : \mathcal{S} \to \mathcal{A}$, such that, over a trajectory $s_0, a_0, s_1, a_1, \ldots$ where $a_i \sim \pi(s_i)$ and $s_{i+1} = \mathcal{T}(s_i, a_i)$ for $i \geq 0$, the total reward is maximized:

$$\max_{\pi} \mathbb{E}\Big[\sum_i \mathcal{R}(s_i, a_i)\Big]. \tag{2}$$

There are potentially many MDPs that could be constructed to help solve the optimized compilation problem. We propose an MDP which possesses the following desirable properties: 1) it is finite, 2) it has a single absorbing state, and 3) all sequences of actions eventually lead to that absorbing state. In essence, we are solving a *stochastic shortest path* problem [6], albeit with an enormous state space.

C Full description of Code Generation MDP

In this section, we provide the full description of the code generation MDP.

C.1 Code generation MDP state

Formally, our MDP is defined in terms of the execution of an abstract stack machine. Given a circuit $S = (V, E)$, where V denotes the gates (or vertices) of the circuit, and E denotes the edges of the circuit, the state space of the *code generation MDP* is given by $\mathcal{S} = \{0, 1\}^{|V|} \times 2^V$. We denote the state of the computation of the circuit as a tuple (p, t). The first component of the state, p, is a boolean value associated with each gate in S, representing whether the gate has been computed, and is used to ensure that the MDP steadily makes forward progress towards obtaining a valid program. The second component of the state, t, is a set of gates in S, and represents values currently available for use by potential instructions.

C.2 Code generation MDP action and transitions

Our MDP is defined to have three actions for each gate in V: *load*, *execute* and *store*. In order to obtain the desired properties of obtaining a valid program, available actions depend on the current state (p, t).

load A load action for gate $v \in V$ is available if $p_v = 1$, $v \notin t$ and $\exists u \in V$ s.t $p_u = 0$ and $(v, u) \in E$. Executing the action will cause the transition $(p, t) \mapsto (p, t \cup \{v\})$.

store A store action for gate $v \in V$ is available if $v \in t$. Executing the action will cause the transition $(p, t) \mapsto (p, t \setminus \{v\})$. Note that if actually storing the value is not necessary, we call this action *drop* instead. It has the same semantics.

execute An execute action for gate $v \in V$ is available if $p_v = 0$, and for all $u \in V$ such that $(u, v) \in E$, we have $u \in t$. Executing the action will cause the transition $(p, t) \mapsto (p', t \cup \{v\})$, where $p'_w = p_w$ for all $w \neq v$, and $p'_v = 1$.

We have illustrated one simplified example in fig. 2. The rules above ensure that any sequence of actions in the code generation 1) eventually terminates, and 2) generates a valid program at termination. Note that in practice, we use a slightly modified version of the MDP (see next appendix C.3) to enable the MDP to reorder operations (e.g., addition under *fast-math* semantics).

C.3 Operation reordering

In some cases, being able to reorder operations may be important to optimize performance. For example, computing $a + b + c + d$ as $(a + b) + (c + d)$ rather than $((a + b) + c) + d$ shortens

[1] In general, $\mathcal{T}$ may be a random function, however, we will only consider deterministic MDPs in this paper.

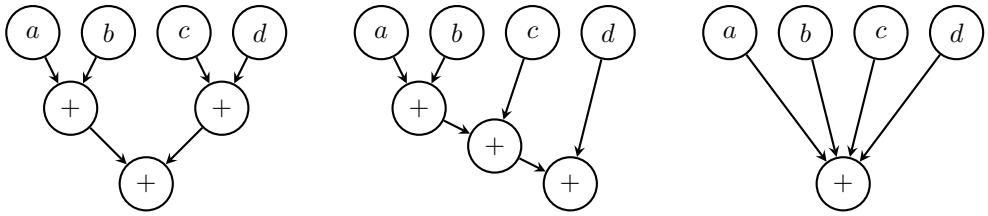


Figure 3: Two "equivalent" circuits for computing $a + b + c = d$, and its multi-arity representation.

the dependency chain, and may improve total latency of the operation. Doing so may sometimes change the output of the program — in particular, these two orders do not produce the same result for all IEEE-754 floating point inputs. Nonetheless, it is helpful to support such semantics, often called *fast-math*, when desired. To encode such semantics, we introduce multi-arity gates, which may take an arbitrary number of inputs, and reduce them according to the given operation (see fig. 3 for an illustration). Unlike other gates in the circuit, these multi-arity gates do not necessarily have a corresponding hardware instruction, and instead must be computed piecemeal. Additionally, it is not necessary (and may be detrimental to performance) to compute all parts of the multi-gate at once.

Given the above two constraints, we transform our circuit in order to better represent multi-gates. The transformation may be given as follows: given a circuit (V, E), and a multi-gate $u \in V$, we modify the circuit as follows: for each incoming edge (s, u), we add a new gate $v_{su} \in V$, an edge (s, v_{su}) and an edge (v_{su}, u). We then remove the edge (s, u).

- The introduced dummy nodes receive special semantics for execute, and cannot be loaded or stored. If no other dummy nodes corresponding to the same multi-operation has been computed, a dummy node may be computed if its input is present in the stack. Otherwise, it may only be computed if both its input and its output (representing the accumulation target) are present in the stack.
- The new changed multi-operation node is modified so that executing the last dummy node marks it as executed. As soon as the first dummy node is executed, this multi-node may be loaded or stored (subject to existing restrictions), representing the loading and storing of the partial accumulation result.

C.4 MDP Termination

One helpful property of our MDP is that it is guaranteed to terminate in finite time, and in a state such that all gates have been computed. We formally state and prove these properties in this sub-section.

Theorem 1. *We consider the code generation MDP. Let s_f denote the state where all gates are computed, and the stack is empty. Consider a sequence of actions $(a_1, \dots)$ and corresponding states $(s_1, \dots)$. Then, (a_i) must be finite, $(a_i) = (a_1, \dots, a_N)$, and $s_N = s_f$.*

Proof. Let x_i denote the number of not computed gates at step i. Note that x_i is a non-negative integer by definition. We show that if $x_i > 0$, then $x_{i+3|V|+1} < x_i$, which ensures that the process terminates by the well-ordering of non-negative integers. Note that as $x_{i+1} \geq x_i$ by definition of the process, it suffices to show that $x_{i+1} \neq x_i$.

Indeed, consider the array $M^i \in \{0, 1, 2\}^{|V|}$ of states for each node, where $M^i_j = 0$ denotes that gate i is loaded at state s_j, $M^i_j = 1$ denotes that it is stored, and $S^i_j = 2$ denotes that it has been reloaded. By the restrictions placed on the actions, we have that if a_i is not an execute action, then $S^i_j \geq S^{i+1}_j$ for all j, and $S^i_{j'} < S^{i+1}_{j'}$ for some j'. We thus deduce that $S^i < S^{i+1}$ in the partial pointwise order on tuples.

By contradiction, assume now that $x_{i+3|V|+1} = x_i$. We must thus have that $a_{i+1}, \dots, a_{i+3|V|+1}$ are not execute actions, and hence we have that $S^{i+1} < \dots < S^{i+3|V|+1}$ in the partial pointwise order on tuples. However, the longest ascending chain in this order has length $3|V|$, a clear contradiction.

To finish, we establish that s_f is the only state with no available actions. We distinguish two cases:

```
load y              load %1, [y]
load x              load %2, [x]
compute xy          mul %3, %1, %2   ; the correct instruction is selected
                                     ;   according to the graph
compute xy + x      store %1         ; no free registers for xy + x,
                                     ;   automatically spill first register
                    add %1, %3, %2   ; now compute xy + x
store xy + x        store %1
```

Figure 4: Example of greedy register assignment for assembly generation on a machine with 3 registers.

- If all gates have been computed, then all values in the stack must have been used. They can thus all be stored. Hence there can only be no store actions available if the stack is empty.
- If there exists a gate which has not been computed, either 1) its compute action is available, or 2) it is missing one of its dependencies, and this dependency may be loaded, or 3) one of its dependency has not been computed. Note that in the third case, we may consider that dependency which has not been computed. As the circuit is finite acyclic, there are no infinite chains of not yet computed gates.

□

D Details of lowering operation

This section contains additional details concerning the fashion in which hardware instructions are produced given a sequence of actions in the MDP defined by the abstract stack machine.

D.1 Register Allocation

We allocate registers using a greedy least-recently-used strategy. Whenever a value is loaded, it is placed in a free register. If none are available, the least recently used register is automatically spilled to main memory. If a spilled value is referenced by an action, it is automatically reloaded into a free register. This process is illustrated in fig. 4. Note that this register allocation strategy is not optimal, nor is it intended to be. By carefully selecting a sequence of actions in the MDP, it is possible to implicitly control register allocation, and we intend this optimization to be carried out by the optimization in the MDP.

D.2 Extending the set of instructions through temporal abstraction

On many target platforms, there may be more than a single instruction which maps to the computation of a given gate (or combination thereof). For example, many floating point units support a fused multiply-add operation, which executes the operation $ab + c$ in a single instruction. In such cases, selecting the correct instruction among the many equivalent ones may be crucial for the performance of the program. One possible approach to enable the selection of alternative instruction would be to encode them as alternatives into the circuit. However, this is prone to combinatorial explosion of the number of alternatives, posing computational problems. We choose to encode such alternatives in a temporal fashion, through the use of options [28], in a process which may be likened to macro-op fusion. We consider a set of templates which may replace exact sequences of actions or instructions to encode alternatives. These substitutions are constrained to be logically equivalent to ensure the validity of the corresponding program. Although similar to the concept of peephole optimization in compiler middle-end, the goal here is not to find all (or even a large portion of) instances where such transformations may be beneficial, but rather to expose new instructions through a specific temporal coding. We expect the policy of the MDP to leverage these transformations where they would be beneficial. We use substitutions to implement in-place operations, fused multiply-add and handling of memory operands on x86 architectures: this process is illustrated in fig. 5 and fully described in appendix F.

```
load y
load x
compute xy
drop y
```

(a_1) Original actions

```
load %1, [y]
load %2, [x]
mul %3, %1, %2
; drop is no-op
```

(a_2) Initial instructions

```
load %1, [y]
load %2, [x]
mul %1, %1, %2
; mul overwrites y
```

(a_3) Post-substitution instructions

(a) In-place operation substitution

```
load y
load x
compute xy
compute xy + x
```

(b_1) Original actions

```
load %1, [y]
load %2, [x]
mul %3, %1, %2
add %4, %2, %3
```

(b_2) Initial instructions

```
load %1, [y]
load %2, [x]
; mul + add is fused
fma %3, %1, %2, %2
```

(b_3) Post-substitution instructions

(b) Fused multiply-add substitution

Figure 5: Example of substitutions applied to actions

E Reward function

Formally, we design the reward as $\mathcal{R} = \mathcal{R}_{\text{cost}} + \mathcal{R}_{\text{progress}}$, where $\mathcal{R}_{\text{cost}}$ captures the cost optimization, and $\mathcal{R}_{\text{progress}}$ is an auxiliary reward designed to encourage the agent to progress.

Progress reward The progress reward $\mathcal{R}_{\text{progress}}$ is an auxiliary reward designed to facilitate training; it rewards the agent for computing gates in the circuit. Formally, we write:

$$\mathcal{R}_{\text{progress}}(s, a) = \sum_{v \in V} p'_v - p_v, \tag{3}$$

where $s = (p, t)$ and $\mathcal{T}(s, a) = (p', t')$. We note that as any full episode $(a_1, \ldots, a_T)$ must compute every gate, we have that for any such episode, $\sum_i \mathcal{R}_{\text{progress}}(s_i, a_i) = |V|$. In particular, in the absence of discounting factor, this term simply adds a constant to the total objective, and does not affect the true optimum. Nonetheless, given the combinatorial nature of the optimization, this term greatly affects the heuristics of the tree search, and thus the results obtained in practice.

Cost reward Given a sequence of actions $(a_1, \ldots, a_T)$, we may consider the resulting program $P_T = P(a_1, \ldots, a_T)$. Note that if the sequence of actions does not reach the terminal state, then P_T is not necessarily a valid program for S. However, we may still evaluate its cost $C(P_T)$. We thus define the cost reward $\mathcal{R}_{\text{cost}}$ such that, for all sequences $(a_1, \ldots, a_T)$:

$$\sum_{i=1}^{T} \mathcal{R}_{\text{cost}}(s_i, a_i) = -C(P(a_1, \ldots, a_T)). \tag{4}$$

Note that depending on the structure of C, we may be required to modify our MDP to include a history state in order for $\mathcal{R}_{\text{cost}}$ to be well defined.

There are two common targets in the context of compute programs: optimizing for size, and optimizing for performance. Although measuring the size of a program is straightforward, estimating its performance may be difficult on modern out-of-order processors, and is an active area of research [14, 1]. We demonstrate how such tools may be integrated into the learning process in our implementation.

F Substitutions

We make use of a form of macro-coding or substitution to implement a wider variety of instructions and behaviors from a baseline streamlined semantic graph. The policy model does not explicitly encode these substitutions except through their interaction with the loss function.

Store to drop Our action space contains three actions (*load*, *store* and *execute*) for each gate in the circuit. The *store* action serves two purposes: 1) it enables the MDP to implicitly control register allocation by freeing the register currently containing the stored gate, and 2) it enables storing the

result to either the output or a temporary location as needed. In some cases, based on the state of the graph, we may determine that a *store* operation can be logically replaced by a no-op, which we call *drop*, and which simply signals that the register is to be freed without performing any instructions. We perform this replacement when 1) the gate has already been stored once in the past, or 2) all successors of the gate have already been computed, and the gate is not an output gate.

Fused Multiply-Add Although our action space does not contain an instruction which directly models a fused multiply-add (fma) instruction, such instructions are important to achieve best performance in numerical code on x86 platforms. We introduce such instructions through a "macro-coding" or "option", given by sequences of the form: `mul r1, r2, r3`; `add r1, r4, r1`. Such sequences are replaced with the corresponding three-operator FMA instruction. We also perform substitutions for various combinations of subtraction and negation to emit fused (negate) multiply add / sub.

Memory operand On x86, the instruction encoding supports the use of a memory operand in most arithmetic instructions. Such an instruction, instead of referencing a register as an input to the operation, instead references a memory location. Supporting such instructions may be useful to reduce code size or optimize the usage of registers. We replace sequences of the form `load r1, [a]`; `op r2, r3, r1`; `drop r1` to make use of the instruction with a memory operand `op r2, r3, [a]`. Note that such instructions do not exist on Arm, and this replacement is thus not performed for our Arm experiments.

G Neural-guided MCTS

We describe the main elements of our implementation in this section, and refer the reader to appendix H for the full details.

G.1 Monte Carlo tree search

Monte Carlo tree search (MCTS) is a widely used class of algorithms for planning in MDPs. An agent employing MCTS for decision making estimates the value of taking actions from its current state by simulating trajectories in an MDP. Specifically, the agent progressively builds an internal model of the environment, represented as a tree in which nodes correspond to states s_i and edges correspond to actions a_i. Each node maintains the current estimated value of the state (i.e., the value obtained by the agent if it continues executing its policy from the state) as well as the number of times the state has been visited during simulation. One step of MCTS in a given state corresponds to a sequence of algorithmic steps in the current internal search tree maintained by the agent:

1. **Selection**: Nodes in the search tree are selected until a leaf node is reached, trading off exploration and exploitation using some variant of the upper confidence bound for trees (UCT) formula.
2. **Expansion**: The leaf node is expanded by adding its children to the search tree.
3. **Simulation**: A rollout is performed from the leaf to a terminal node using a random policy.
4. **Back-propagation**: The value of the simulated trajectory is propagated back to the nodes in the agent's search tree. The number of visits to each node traversed in the selection step is also incremented.

After some number of steps, the agent executes an action according to the visit counts, and begins the deliberation steps listed above from the next state, and so on until a terminal state is reached.

G.2 Neural-guided MCTS

MCTS can be augmented with a neural network f_θ which aids in the search process. f_θ takes as input a state $\mathcal{S}$ and outputs a tuple (π_S, v_S), where $\pi_\mathcal{S}$ is a policy over actions from state $\mathcal{S}$ and $v_\mathcal{S}$ is the estimated value of state $\mathcal{S}$. $\pi_\mathcal{S}$ aims to match the policy obtained in MCTS, as defined by the number of visits to each child node from a state, while $v_\mathcal{S}$ aims to match the true value observed from a given state. The policy is then used as a prior in the selection step of MCTS, and the estimated

value predicted by the network is used instead of doing full simulations to terminal states in the simulation step. The neural network is continuously improved by training on data from trajectories in the MDP obtained through MCTS. Note that in such a system, each expansion / simulation step requires an evaluation of a neural network. To increase the throughput of evaluations, we search the tree in parallel by using the watch-unobserved formulation [15].

G.3 Neural network implementation

To implement our neural guided MCTS, we must parametrize the policy-value function, that is, a function which for each state S of the MDP associates a vector of probabilities π_S and a real number v_S. A state in our code generation MDP is naturally represented by a graph: the current state of the computation graph of the target program. This motivates the use of a graph neural network for the policy value function. Given a history of length h of states $(S_{i-h}, \ldots, S_i)$, we generate a node-level embedding for each gate at each state, combine the embeddings across the history to obtain a state S_i^h, and use this as input to a message-passing graph neural network which outputs $\pi_{S_i^h}$ and $v_{S_i^h}$.

G.4 Training process

The neural network f_θ is trained by cycling between two steps: 1) generating trajectories using MCTS guided by the current network, and 2) training on data obtained in these trajectories to update the network parameters θ. A trajectory of length T consists of a sequence of transition tuples $\{(s, n, r, v)_i\}_{i=1}^T$ where s_i is the state of the computation graph at step i, n_i is the number of visits to each of the possible next actions from state s_i, r_i is the reward associated with the transition from s_i to s_{i+1}, and v_i is the value observed in the trajectory from state s_i. Note that the value v_i associated with a state s_i is equal to $v_i = \sum_{j=i}^T r_j$. For a given transition tuple $(s, n, r, v)_i$, the loss for f_θ is:

$$\mathcal{L} = (v_{s_i^h} - v_i)^2 - n_i^T \log(\pi_{s_i^h}). \tag{5}$$

After generating some number of trajectories with f_θ held fixed, f_θ is updated by stochastic gradient descent to minimize Eq. 5 over mini-batches of transition tuples from the generated trajectories.

Note that the overall implementation of the distributed MCTS and neural network training contains many details and hyper-parameters, which we have collected in appendix H.

H Technical overview of implementation

In order to support the scale required, we implement our described neural-guided MCTS using the distributed computing platform ray. There are two main components to the system: the rollout workers, which execute MCTS rollouts according to a given neural network guiding the policy, and the gradient trainer, which takes MCTS rollouts and fits the network to the observed transition statistics.

H.1 Overview of system architecture

Our system consists of a set of rollout workers, and gradient trainers which communicate through a replay buffer. Additionally, rollout workers need to evaluate the policy model, which is much more efficient on GPU, which leads us to implement a client-server architecture for policy evaluation. In order to achieve good performance for GPU evaluation of the network, larger batch sizes are crucial. We thus implement an asynchronous rollout system, where workers perform multiple rollouts simultaneously, queueing up policy evaluation requests and sending them in batches to an evaluation server. A schema of our high-level architecture is provided in fig. 6.

H.2 Policy-value network design

Our method is underpinned by a network which, given a state s, computes an estimate for the current value v_s and a policy (distribution over actions) π^s. We fully describe the implementation of the network in this section.

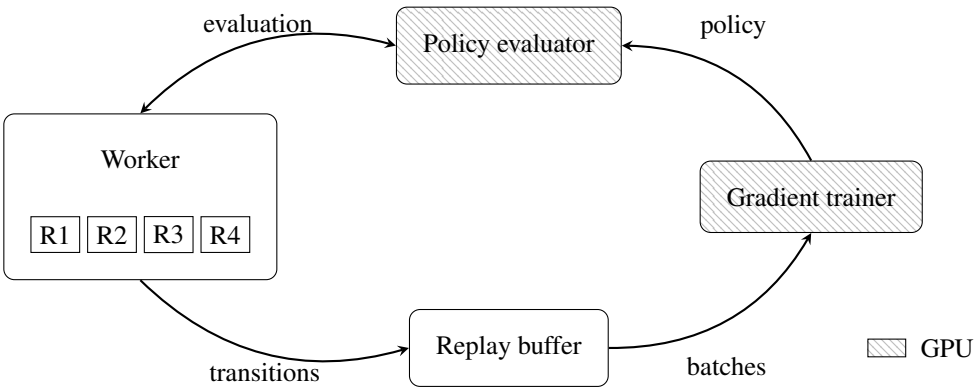


Figure 6: Schema of system components and communication.

Message passing neural networks Given the representation of the environment as a circuit (i.e. a graph), it is natural to design the policy value function as a graph neural network. We design our policy-value network as plain message passing neural network (MPNN), with mean and max aggregation. More formally, let $x_i^{(l)} \in \mathbb{R}^h$ denote the value of the embedding at layer l and gate i, then the network computes:

$$\begin{aligned} m_{i\to j}^{(l)} &= (x_i^{(l)}, x_j^{(l)}, e_{i\to j}) W_s^l, \\ y_j^{(l)} &= \big(\frac{1}{d_{\text{in}}(j)} \sum_{i:(i,j)\in E} m_{i\to j}^{(l)}, \max_{i:(i,j)\in E} m_{i\to j}^{(l)}\big), \\ x_i^{(l+1)} &= \text{GRU}(y_i^l, x_i^l), \end{aligned}$$

where $W_s^l \in \mathbb{R}^{2h+h_e,h}, W_r^l \in \mathbb{R}^{2h,h}$ denote learnable weight matrices, $(\cdot,\cdot,\dots)$ denotes concatenation of vectors, $e_{i\to j} \in \mathbb{R}^{h_e}$ denotes a feature vector associated with the edge $i \to j$, $\max$ is understood to denote pointwise maximum of vectors, and $d_{\text{in}}(j)$ denotes the in-degree of node j.

In all experiments, we set the number of layers to 4, and the number of hidden units to 128.

Graph featurization In order to leverage our GNN architecture, we featurize the state of the MDP as a graph with node and edge features. We modify the graph derived from the circuit in the following fashion: 1) we add an additional "virtual node", which is connected to every gate in the circuit, and 2) for every edge (i, j), we associate a learnable edge feature $e^{\text{in}} \in \mathbb{R}^{h_e}$, and add an edge (j, i) with learnable edge feature e^{out}.

We featurize the state of every node by a learnable embedding. In order to model history-dependent effects, we concatenate such a representation for a fixed number of history state in order to produce the initial node embedding. Additionally, we featurize the current position of the node in the stack of the abstract machine as a discrete quantity, truncated at a fixed depth.

In our experiments, we truncate the node stack information at 4, and accumulate 8 total history states.

Policy readout The un-normalized policy prediction is obtained by applying a MLP to the embedding of each node in the GNN, producing three outputs per node, corresponding to the un-normalized logits of the load, store and execute actions. Prior to normalization, predictions for actions invalid in the current state are masked by setting their logit to $-\infty$. In our experiments, we use a two-layer MLP with ReLU nonlinearity with 128 hidden units.

Value readout and linear value predictor Our environment is particular in that the value is approximately linear as a function of the number of gates not computed, representing to first approximation the total cost of performing all operations. We take advantage of this by parametrizing our value predictor as:

$$\hat{v} = \hat{\beta} x + \hat{f}(s),$$

where x denotes the number of gates not computed, $\hat{\beta} \in \mathbb{R}$ denotes a scalar coefficient, and $\hat{f}$ denotes the neural network. This facilitates training by ensuring that the neural network learns a more stationary quantity. The neural part of the value predictor is obtained by applying a MLP to the embedding of the virtual node. In our experiments, we use a two-layer MLP with ReLU nonlinearity with 128 hidden units.

H.3 Rollout

The rollout workers execute the MCTS steps, and produce transitions according to the current policy. We implement MCTS according in a fashion similar to [25].

MCTS Selection Our selection rule is given by selecting the node with the maximal score:

$$\text{score}_i = v_i + \pi_i \frac{\sqrt{N}}{1 + N_i} \Big(c_1 + \log\big(\frac{1 + N + c_2}{c_2}\big) \Big), \tag{6}$$

where N_i is the visit count for child i, $N = \sum_i N_i$ the total visit count of the parent, v_i the current estimate of the value of child i, and π_i the prior for child i as evaluated by the current neural policy. Due to the potential large range of values in our MDP, we normalize v_i so that it takes values in $[0, 1]$ by linearly rescaling using the largest and smallest value of nodes currently in the tree. In order to support our asynchronous policy evaluation, we enable parallel selections in the tree using the watch-unobserved formalism [15]. The visit counts N_i are interpreted as total visit counts (including unobserved visits), whereas v_i is the current estimate of the value based on observed visits. When there are no observed visits for child i, we set its value $v_i = 0$.

Exploration noise In order to encourage exploration when generating rollouts, the prior policy π produced by the neural network is perturbed when initializing a new node. This perturbed prior $\tilde{\pi}$ is obtained by mixing Dirichlet noise in the following fashion:

$$\tilde{\pi}_i = (1 - q)\pi_i + q n_i, \tag{7}$$

where $(n_1, \ldots, n_k) \sim \text{Dirichlet}(\alpha)$. In all our experiments, we use $q = 0.25$ and $\alpha = 0.25$.

Self-play We generate sequences of transitions through self-play. We investigate two self-play strategies: 1) full rollouts, and 2) episode rollouts.

Full rollout The problem is initialized from the initial state of the MDP. At each step, a fixed number of MCTS selections are performed, after which the next action is selected with probability proportional to their visit count. This process is repeated until the terminal state of the MDP is reached.

Episode rollout A first play through is performed from the initial state to the final state by sampling actions directly according to the prior policy π without MCTS. Among these steps, one is chosen uniformly at random to serve as the initial state for the rollout. Self-play with full MCTS is then performed for a fixed number of steps. Finally, the rollout is finished after those steps by again sampling from the prior policy to the final state.

We found that generally, joint learning with full rollouts lead to better final solutions. However, it leads in much slower training, as the latency of a single generation is bound by the total time required to traverse the MDP, which may be significant for larger problems. In all cases, we used 50 simulations per step.

H.4 Gradient training

We implement the policy and value function as a graph neural network, and train it by gradient descent on the generated transitions.

Loss We train our model to match the computed value and policy through gradient descent on the loss. We compute our loss as the weighted sum of two components:

$$\mathcal{L} = \mathcal{L}_{\text{policy}} + \lambda \mathcal{L}_{\text{value}}.$$

We choose $\mathcal{L}_{\text{policy}}$ to be the cross-entropy loss between the empirical distribution of the sampled actions and the predicted policy, and $\mathcal{L}_{\text{value}}$ to be a "smooth L1" loss on the estimated value:

$$\mathcal{L}_{\text{value}}(v, \hat{v}) = \begin{cases} (v - \hat{v})^2 & \text{if } |v - \hat{v}| < \gamma, \\ \gamma |v - \hat{v}| & \text{otherwise.} \end{cases} \tag{8}$$

We choose $\lambda = 0.1$ and $\gamma = 10$ for all problems.

Updating the rollout policy After training for a certain amount of time, the gradient trainer must update the policy used in the rollouts. We consider two strategies: 1) generational training, where we alternate fixing the policy to generate rollouts, and training the policy on the generated rollouts, and 2) continuous training, where the policy is (semi-)continuously updated. We found that generational training with full rollouts performed best on small scale problems, eventually finding more optimal solutions. However, this procedure is challenging to scale due to the required time for each generation, and we instead use of continuous training. We update the policy used by the rollout workers for the first time after gradient descent on 250000 transitions has been performed. Subsequently, the number of observed transitions between updates is increased by a factor of 1.2, up to a maximum of $10\times$ the original number.

Model weights and warm starting We generally found that maintaining model weights for extended amounts of time without resetting the model lead to suboptimal performance, with the behavior suggesting that the model was stuck in local optima. To prevent this issue, we re-initialize the model weights periodically. In order to ensure that the model uploaded to the rollout workers is of good quality, we require a certain number of gradient steps before a re-initialized model may be uploaded. We perform this by training two models in parallels: 1) a main model and 2) a backup model. At every upload cycle, the main model is uploaded to the rollout workers, then replaced with the backup model. The backup model is then re-initialized using random initialization. These two models are otherwise trained on the same data.

Optimizer and learning rate We perform the optimization using the `Adam` optimizer with decoupled weight decay, with an initial learning rate of 5×10^{-4} and a weight decay of 10^{-5}. We apply gradient clipping by norm with a clip value of 1.0. The learning rate is annealed using a cosine annealing schedule, which is reset after every model upload. Additionally, we reset all optimizer state (i.e. collected gradient moments for Adam) after each model upload, for both the main and the backup model.

Linear value predictor The linear value predictor is not trained through gradient training, but rather through online least-squares with an exponentially weighted moving average. More precisely, let $x_t \in \mathbb{R}$ denote the input feature and $y_t \in \mathbb{R}$ the target value of a new sample, then the coefficient of the linear value estimator is updated as:

$$\beta^{t+1} = (1 - \nu)\beta^t + \nu \frac{y_t}{x_t}. \tag{9}$$

In all experiments, we use $\nu = 0.05$.

Transition sampling The rollout workers store the obtained transitions in a replay buffer. Samples for training are obtained by sampling uniformly at random at each step among those in the replay buffer, with no prioritization. The replay buffer size used for all problems is 100000 transitions. The observed replay ratio (i.e. the average number of times a transition is used for training before being discarded from the replay buffer) ranges from 5 to 100 depending on the problem. We made no attempt to tightly control the replay ratio.

H.5 Cost model

Our training process requires a cost model to estimate the performance of the model so far. We implement three different cost models for different targets: 1) Arm Cortex M-4 performance, 2) x86 Intel performance, and 3) x86 code size. We describe the implementation detail for each of these

```python
def arm_m4_cost(ops: List[Operation]):
    total_cycles = 0
    used_registers = set()

    last_output: int = None

    for op in ops:
        # Get the number of cycles for the given operation
        # as specified by the Cortex M-4 reference manual
        total_cycles += cycles_for_op(op)

        # The cortex m4 FPU has an additional cycle penalty
        # when directly reusing the output of the previous operation.
        if last_output in op.inputs:
            total_cycles += 1
        last_output = op.output
        used_registers.add(last_output)

    # The Arm64 calling convention requires the callee to preserve
    # registers 16-32. If we happen to use these registers, we must
    # save them to, and restore them from the stack.
    # We take into account the cost here
    num_registers_to_preserve = len([r for r in used_registers if r >= 16])
    if num_registers_to_preserve > 0:
        total_cycles += 2 + 2 * num_registers_to_preserve

    return total_cycles
```

Listing 1: Pseudo-code for cost model on the Cortex M-4 platform.

Arm Cortex M-4 We construct a performance model for the Arm Cortex-M4 FPU based on the instruction throughput provided in [3]. We additionally model the FPU dependency penalty when an instruction immediately consumes the output of a floating point arithmetic instruction. Finally, we model the cost of spilling and restoring registers which must be preserved according to the Arm64 calling convention. We do not attempt to model the out-of-order characteristics of the `sqrt` or `div` instructions. A pseudo-code implementation is provided listing 1.

x86 Intel performance We make use of OSACA [14] to estimate performance for Intel CPUs. Our generated program is translated on the fly to x86 assembly, and its performance is estimated through OSACA. We note that we estimate throughput with fixed port utilization (rather than optimal planning) as this cost model is already particularly expensive to evaluate due to its complex nature. Estimates in the paper are reported with optimal port utilization. By default, the OSACA software produces a (reverse) throughput metric for each port in the processor (see listing 2 for a sample output from the OSACA tool). We summarize this metric into a single scalar by taking the maximal reverse throughput as the estimated number of cycles per iteration. In the tuned version, we introduce a hyper-parameter α which controls the aggregation between the different ports: let t_i denote the reverse throughput (as estimated by OSACA) for port i, and suppose that there are p ports, then we consider the α-mean:

$$C_\alpha(P) = \left(\frac{1}{p}\sum_{i=1}^{p} t_i^\alpha\right)^{1/\alpha}.$$

We use $\alpha = 2$ when using the "tuned" cost.

x86 code size To estimate code size of x86 code, we assemble the generated program into x86 machine code on the fly using the xbyak (`https://github.com/herumi/xbyak`) library. We then measure the generated code size in bytes.

```
Combined Analysis Report
------------------------
                                    Port pressure in cycles
     |  0   - ODV  |  1   |  2   -  2D  |  3   -  3D  |  4   |  5   |  6   |  7   ||  CP  | LCD  |
-----------------------------------------------------------------------------------------------------
   2 |             |      | 0.50   0.50 | 0.50   0.50 |      |      |      |      ||  4.0 |      |   vmovss (%rdi),%xmm0
   3 |             |      | 0.50   0.50 | 0.50   0.50 |      |      |      |      ||      |      |   vmovss 0x4(%rdi),%xmm1
   4 | 0.50        | 0.50 |             |             |      |      |      |      ||  4.0 |      |   vmulss %xmm1,%xmm0,%xmm2
   5 | 0.50        | 0.50 |             |             |      |      |      |      ||  4.0 |      |   vfmadd213ss %xmm1,%xmm0,%xmm2
   6 |             |      | 0.00        | 0.00        | 1.00 |      |      | 1.00 ||  0.0 |      |   vmovss %xmm2,0x8(%rdi)

       1.00         1.00   1.00   1.00   1.00   1.00   1.00                1.00      12    0.0
```

Listing 2: Sample output from OSACA [14].

I Sample problem descriptions

We evaluate our method on a number of short computationally intensive tasks. We describe the tasks in this section, including some background on their applications, and comment on the corresponding graph.

I.1 Cholesky decomposition

The Cholesky decomposition is a central tool in numerical linear algebra. For a positive definite matrix Σ, it aims to find a lower-triangular matrix L such that $\Sigma = LL^\top$. Such a factorization is often a precursor for solving a linear system, or obtaining other characteristics of the matrix Σ. We implement a static version of the Cholesky-Banachiewisz algorithm on a 4×4 matrix.

I.2 Matrix multiplication (GEMM)

Multiplication of general matrices (GEMM) is perhaps the most important linear algebra subroutine. It consists of computing the output $C = AB$ for general (unstructured) matrices A and B. We consider a variant of the problem where B has known (fixed) structural sparsity, so that the value is only specified for some entries of B (known at compilation time), and the remaining entries are assumed to be zero. Such a subroutine is central in many scientific applications, and also for neural network inference, were compression techniques may enable one to sparsify the weight matrix of the neural network. We fix the sparsity pattern to a randomly generated pattern with 50% sparsity, constrained such that no entries of C are structurally sparse.

I.3 Gauss-Seidel Iteration

The Gauss-Seidel method is an iterative method for solving linear system. Given a matrix $A = L_\star + U$, where $L_\star$ is strictly lower triangular and U is upper triangular, it computes the iterate:

$$x^{t+1} = L_\star^{-1}(b - Ux^t). \tag{10}$$

Under certain conditions on A, such iterates converge to the solution of the linear system $Ax = b$. We implement a single such iteration with structural sparsity assumptions on A.

I.4 Energy of a neo-Hookean solid

A neo-Hookean is a model used to predict the stress-strain behavior of materials under deformations. Its value for a 3-dimensional problem is given by:

$$E = \frac{C_1}{2}(I_1 - 3 - 2\log J) + \frac{D_1}{2}(J-1)^2, \tag{11}$$

where $I_1 = \mathrm{tr}(F^\top F)$ and $J = \det F$, and F denotes the 3×3 deformation gradient. As most processors do not have native instructions to compute the logarithm, we have replaced it with a 5th order polynomial approximation in all cases.

I.5 Runge-Kutta Integration Step

The Runge-Kutta family of ODE integrators is one of the most widely used integrator for solving initial value problems in ordinary differential equations. For a initial value problem $\dot{y} = f(t, y)$ with

a step size of h, the 4th order integrator, often referred to as RK4, is given by:

$$y_{t+h} = y_t + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{12}$$

$$k_1 = f(t, y_t), \tag{13}$$

$$k_2 = f(t + \frac{h}{2}, y + \frac{h}{2}k_1), \tag{14}$$

$$k_3 = f(t + \frac{h}{2}, y + \frac{h}{2}k_2), \tag{15}$$

$$k_4 = f(t + h, y + hk_3). \tag{16}$$

We implement a single iteration of a RK4 integrator (computing y_{t+h} from y_t) for the problem $f(t, y) = yt - (yt)^2$.

J Discussion of results

On the ARM Cortex-M4 platform, we systematically outperform gcc as shown in table 1. This is due to a combination of factors. 1) on the Cortex-M4 platform, it is not always beneficial to perform fused multiply-add operations, as they incur an implicit one cycle penalty due to a dependency. Through the feedback from the cost model, our model may automatically decide whether to use the fused multiply-add operation, or to modify the scheduling to minimized dependencies between successive instructions. 2) Due to the ARM ABI, the first 16 floating point registers may be freely used. However, functions must preserve the upper 16 floating point registers (out of 32), and using those requires spilling and restoring them. Our model is automatically able to balance the freedom of additional registers vs the cost of spilling and restoring them as it is reflected in the cost model.

On the other hand, we see in table 2 that on x86, our performance is mixed. We note that, as we are directly optimizing the estimator, the generated programs become adversarial towards our model (compared to the programs generated by gcc): the estimator is overly optimistic for our programs compared to those generated by gcc, leading to suboptimal results. Such phenomenon is also observed when learning physical processes from simulations, and is often called "sim2real". This highlights the importance of accurate simulators, and the possibility to directly learn from empirical data. We are able to partially address this discrepancy by tuning the aggregation of the OASACA cost model (see appendix H.5).

K Details of experiments

K.1 Compiler baselines

In order to produce our compiler baselines, we mechanically translate our computation graph into C code, and compile the resulting C code using a standard C compiler. This translation is performed by creating a single variable for each gate, and assigning the result corresponding to the operation specified by that gate, with each gate processed in an unspecified order compatible with the topological order of the circuit. All operations are done using single precision floating point computation. The signature of the C function takes a single argument, a pointer to an array of floats, which is expected to contain the inputs and constants at pre-determined offsets, and which expects to receive the output of the computation at subsequent offsets. The generated C code and the programs we generate through MCTS are considered equivalent under fast-math semantics (though they may be different under more strict semantics due to e.g. differences in floating-point accumulation order).

For performance comparisons, the compilation is performed using the `-O3` and `-ffast-math` flags in order to ensure equivalent semantics. On x86, we additional specify access to AVX2 instructions and FMA instructions through the `-mavx2` and `-mfma` flags, but we disable automatic vectorization with the `-fno-tree-vectorize` option. On Arm Cortex M-4, we specify access to the hardware FPU with the following flags: `-mcpu=cortex-m4 -mfpu=fpv4-sp-d16 -mfloat-abi=hard`.

For size comparison on x86, the `-O3` flag is replaced with the `-Oz` flag, requesting optimization for code size above all considerations.

Problem	Time (mm:ss)
GEMM $4 \times 4 \times 4$ sparse	23:30
Cholesky 4×4	18:10
Gauss-Seidel 5×5 sparse	2:50
Neo-Hookean energy 3D	31:20
Runge Kutta 4	5:20

Table 4: Time until final kernel is generated

K.2 Empirical data

In order to obtain empirical performance data, we implement benchmarks for the generated functions, both for GCC and our MCTS method.

Arm Cortex-M4 We measure the performance for the Arm Cortex-M4 model on a Arduino 33 BLE based on the nRF52840 microcontroller. Generated functions are separately compiled in their assembly file, and then called within a tight loop of 10000 iterations. The runtime of these iteration (including loop counter comparison and function call) is measured using the internal cycle counter `DWT_CYCCNT`, and communicated through the serial port to a monitoring computer. This process is performed at least 100 times, and the average and standard error of the measured performance is reported.

Intel x86 We measure the performance for the Intel x86 model on a Xeon Gold 6234 CPU clocked at 3.3 Ghz. The generated assembly (for both GCC and our method) is modified by including instructions to record the cycle counter (through the `RDTSC` instruction), as well as loop 10000 times over the original code. This modified function is separately assembled from its assembly source code to prevent compiler optimization through the timing loop into the target code. Each kernel is called 10000 times (for a total of 10^8 iterations), and the average running time (in cycles) and its standard error are reported. The process is pinned to a single cpu core when running the benchmark.

K.3 Hyper-parameter selection

The parameters described in appendices C and H were set using a combination of automatic and manual tuning on training for a $2 \times 2 \times 3$ matrix multiplication problem with instruction count as the target cost function. The number of layers of the GNN, the number of hidden units and the learning rate were tuned automatically using hyper-parameter search to optimize the validation loss on a fixed set of transitions from a $2 \times 2 \times 3$ matrix multiplication problem.

K.4 Training details

We perform training on a distributed system, connecting CPU workers with a GPU server which handles policy evaluation and gradient training. In general, we allocate one GPU for gradient training, one GPU for policy evaluation, and 64 CPU cores for worker rollouts. When training with the OSACA estimator for performance, we instead use 128 CPU cores in order to compensate for the large increase in computational requirements to evaluate the cost function.

For performance, we both train and evaluate the policy neural network in 16-bit precision.

We train until convergence of the average rollout value, and save the best rollout encountered (as determined by the cost function) to be the output of the MCTS code generation procedure. The time until the kernel evaluated in table 2 is produced is listed in table 4. Note that in practice, training is extended significantly beyond that time, as it is impossible to know ahead of time whether a more optimal solution exists. We have included a few training curves in fig. 7 to illustrate the general behavior of the system throughout training.

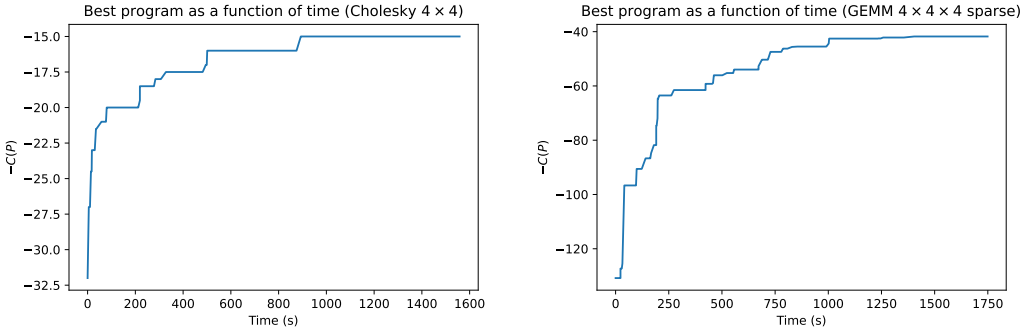


Figure 7: Training progress for selected problems.