ML^{2}Tuner: Efficient Code Tuning via Multi-Level Machine Learning Models

JooHyoung Cha[1] **Munyoung Lee**[2] **Jinse Kwon**[2] **Jubin Lee**[3] **Jemin Lee**[2] **Yongin Kwon**[2*]
[1]University of Science and Technology
[2]Electronics and Telecommunications Research Institute [3]Neubility
{jh.cha, munyounglee, kwonse, leejaymin, yongin.kwon}@etri.re.kr[1,2]
{jubin0927}@gmail.com[3]

Abstract

The increasing complexity of deep learning models necessitates specialized hardware and software optimizations, particularly for deep learning accelerators. Existing autotuning methods often suffer from prolonged tuning times due to profiling invalid configurations, which can cause runtime errors. We introduce ML^{2}Tuner, a multi-level machine learning tuning technique that enhances autotuning efficiency by incorporating a validity prediction model to filter out invalid configurations and an advanced performance prediction model utilizing hidden features from the compilation process. Experimental results on an extended VTA accelerator demonstrate that ML^{2}Tuner achieves equivalent performance improvements using only 12.3% of the samples required with a similar approach as TVM and reduces invalid profiling attempts by an average of 60.8%, Highlighting its potential to enhance autotuning performance by filtering out invalid configurations

1 Introduction

The increasing complexity of deep learning (DL) models has resulted in a significant increase in computational and memory demands [1, 2], prompting growing attention to research on specialized hardware and software optimizations for efficient operation processing within resource constraints [3]. In addition to widely recognized accelerators like Google's TPU [4], Nvidia's Tensor Core[5], and Intel's Gaudi [6], a variety of DL accelerators are under active development. These accelerators are designed to specialize in DL computations, such as matrix-matrix multiplication, but to fully leverage the hardware's parallelism and maximize the use of internal memory, substantial software optimizations are required.

DL accelerator interfaces are typically divided into two types: high-level operator libraries [7, 8, 9, 10, 11] and low-level libraries [12, 13, 14]. High-level operator libraries execute combinations of low-level code based on predefined rules that depend on operation inputs. While this approach is easy to apply, it has inherent optimization limitations, as it cannot define rules for all possible DL operations. In contrast, directly utilizing low-level libraries or generating machine codes can be challenging for users without in-depth hardware knowledge. DL compilers [15, 16, 17, 18] bridge this gap by taking DL operation inputs and autonomously generating optimized code.

Recent research on DL compiler has embraced machine learning-based autotuning techniques bypassing the requirement of domain expertise for reducing development time and the need for extensive human resources [19, 20, 21]. Autotuning compensates for the lack of domain knowledge by exploring numerous code configurations and utilizing performance metrics to train a machine learning model for code optimization.

Template-based methods, such as AutoTVM[22], limit the search space, enhancing the chances of generating valid configurations but constraining the discovery of optimal solutions beyond the

*Corresponding author.

Machine Learning for Systems Workshop at (NeurIPS 2024).

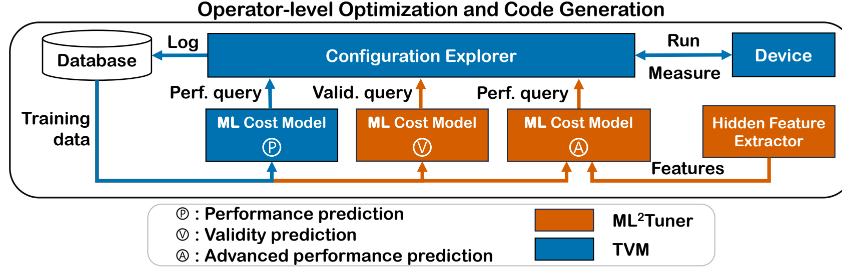

Figure 1: Overview of the automated tuning process of ML^{2}Tuner. Components in blue represent the TVM's tuning process, while ML^{2}Tuner extends the components highlighted in orange.

predefined space. On the other hand, template-free methods such as Ansor [23] investigate a more extensive search space, which expands the possibility of finding better solutions but also increases the risk of generating invalid configurations by considering a wider range of options.

Irrespective of whether a template-based or template-free approach is used, identifying invalid configurations remains a significant challenge. The use of invalid configurations during machine learning model training can lead to reduced efficiency in the autotuning process and degrade the performance of the final optimized code [24, 25]. The use of sophisticated back-end compilers, such as LLVM [26], helps minimize the probability of erroneous or invalid code generation on CPUs and GPUs. However, certain configurations can still lead to the generation of invalid code or sub-optimal performance in LLVM. In contrast, DL accelerators that rely on scratchpad memory and lack the capacity for sophisticated back-end compilers often exhibit erroneous values or encounter runtime errors [27, 28, 29].

To address the issue of prolonged tuning times caused by profiling invalid configurations, we propose ML^{2}Tuner, a multi-level machine learning tuning technique. While existing DL compilers attempt to reduce the number of invalid configurations, the diversity of hardware and architectures still leads to tuning delays and reduced cost model performance [30, 31]. By incorporating an additional validity prediction model focused solely on predicting configuration validity and training the performance prediction model exclusively with valid configurations, we can achieve more accurate performance predictions. Furthermore, by integrating an advanced performance prediction model that considers hidden features generated during the compilation process. Compared to TVM approach, ML^{2}Tuner achieves equivalent performance improvements across all ResNet18 layers while using only 12.3% of the sample size and reducing the average number of invalid profiling attempts by 60.8%.

2 System Design of ML^{2}Tuner

We introduce ML^{2}Tuner, an execution configuration tuner that employs multi-level machine learning, specifically designed for DL accelerators. Fig. 1 illustrates the overall automated tuning process of ML^{2}Tuner. While preserving the core components of TVM's configuration explorer, ML cost model (P), and database, ML^{2}Tuner enhances functionality by incorporating two additional ML cost models (V and A), along with a hidden feature extractor to more effectively identify valid configurations and accurately predict performance.

Configuration Explorer: The configuration explorer selects N code configurations with the highest potential for optimal performance to profile. It begins by defining the search space of all configurations for the given DL operation and assesses their potential based on predictions from the ML models. A code configuration can be a machine code or combination of low-level library calls.

ML Model P: Model P predicts the performance of configurations based on features provided by the configuration explorer, functioning similarly to the model employed in TVM. However, unlike TVM, which directly uses the configurations suggested by this model in subsequent profiling iterations, ML^{2}Tuner further evaluates these configurations using the models V and A.

ML Model V: Model V is dedicated to predicting the validity of configurations, utilizing the same features as Model P. Even if Model P predicts a configuration as highly optimal, ML^{2}Tuner avoids profiling it if Model V predicts it to be invalid. The configuration explorer iteratively applies models

P and V until it accumulates $(\alpha+1)\times N$ configurations. Once enough configurations are collected, ML^{2}Tuner moves on to the next step.

Hidden Feature Extractor & ML Model A: From the $(\alpha+1)\times N$ configurations selected—ranked by the highest predicted performance from Model P—Model A selects the final N configurations expected to yield better performance.

Model A leverages not only the visible features used in Model P's predictions but also internal hidden features generated during the compilation process. During compilation, static analysis and optimization passes produce features such as branch decisions and loop size determinations. By incorporating these hidden features alongside the visible ones, ML^{2}Tuner builds a more precise performance prediction model without requiring extensive domain knowledge for hardware. Consequently, ML^{2}Tuner compiles all $(\alpha+1)\times N$ configurations recommended by models P and V, extracts hidden features during this process, and uses model A to re-evaluate and select the final N configurations.

Profiling & Training: The final configurations selected by the configuration explorer are executed on real hardware. Validity is assessed by checking for crashes and verifying output correctness. For valid configurations, execution times are measured and logged into the database. In each iteration, N configurations are executed; the validity results are used to train Model V, while the execution times of the valid configurations are used to train Models P and A.

3 Experimental Results

Experimental Setup: To validate the effectiveness of ML^{2}Tuner, we extended the open-source DL accelerator VTA [32] on a Xilinx ZCU102 board [33]. We also implemented the core components of ML^{2}Tuner, including the configuration explorer, hidden feature extractor, and the three ML cost models(P, V, and A), in the PyTorch Glow compiler [34] to compare the performance of ML^{2}Tuner with that of TVM, which uses only the ML cost model P. Additionally, we developed low-level libraries [35] for our VTA and implemented a back-end compiler [17] to execute DL operations by integrating these components. For machine learning-based autotuning, ML^{2}Tuner adopts XGBoost (v2.1.1)[36]. To identify and extract hidden features, ML^{2}Tuner uses an internally integrated compiler. It collects data such as iteration counts from configurations, values affected by conditional expressions, and variations resulting from branch statements. Furthermore, it captures details about the optimization and internal tiling strategies during the code generation process.

Our experiments targeted 10 types of convolution layers from ResNet18 [37], trained on the ImageNet dataset, with the hyperparameters set to $N=10$ and $\alpha=1.0$. The details of the experimental setup, such as the hyperparameters, are specified in Appendix A.

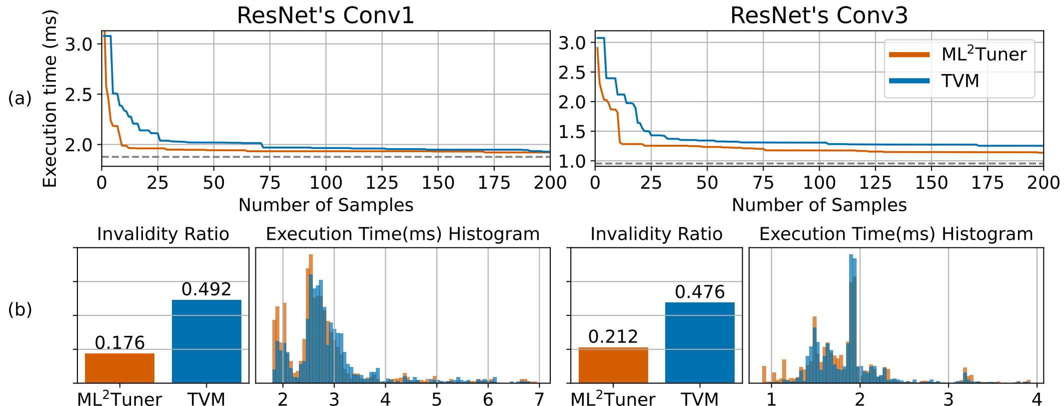

Figure 2: Experimental results for Conv1 and Conv2 of ResNet18. The orange represents the results of ML^{2}Tuner, while the blue represents the results of the TVM approach. (a) The x-axis shows the number of configurations tested during the tuning process, and the y-axis shows the lowest execution time among the cumulative configurations. (b) The left plot displays the invalidity ratio, while the right plot presents a normalized histogram of execution times for the valid configurations.

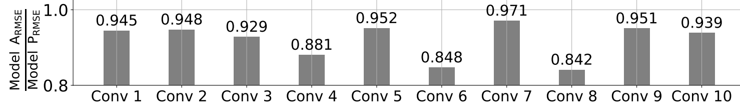

Figure 3: Ratio of RMSE values of model A compared to model P across different layers.

Overall Performance: Fig. 2 (a) compares the tuning process of each ResNet18 layer using ML^{2}Tuner with the existing TVM method that employs a single machine learning model for performance prediction. Due to the random initialization of the machine learning models, results may vary with each attempt; therefore, we conducted 10 experiments and plotted the average values.

In our analysis, we identified the point of convergence in TVM's configuration exploration as the stage where the same value repeated more than 10 times. We then compared the average number of configuration explorations required by ML^{2}Tuner to reach equivalent performance. The results indicate that, for ResNet's Conv1 layer, ML^{2}Tuner achieved equivalent performance using only 11.2% of the configurations attempted by TVM. For the Conv3 layer, this percentage was 11.3%, with an average of 12.3% across all ResNet18 layers.

Distribution of configurations: Fig. 2 (b) shows the invalidity ratio of the configurations proposed by the machine learning models of ML^{2}Tuner and TVM during the tuning process and a normalized performance histogram for valid configurations. In preliminary experiments, random sampling for Conv1 yielded an invalidity ratio of 0.926. Using TVM's model reduced the invalidity ratio to 0.492, indicating a higher rate of selecting valid configurations than random chance. However, ML^{2}Tuner further reduced the invalidity ratio to 0.176, demonstrating greater efficiency in tuning time by avoiding invalid configurations. Similar trends were observed for Conv2 through Conv10. The histogram confirms that ML^{2}Tuner not only identifies valid configurations more effectively but also selects configurations skewed toward better performance, as indicated by the leftward shift.

Impact of Hidden Features: To demonstrate that ML model A predicts performance more accurately than ML model P by utilizing additional hidden features, we measured the Root Mean Square Error (RMSE) of both models. Fig. 3 presents the RMSE ratio of model A compared to model P for each ResNet18 layer. The experimental results show an average ratio of 0.919, indicating that model A achieves lower prediction errors. Although extracting hidden features requires compiling α times more configurations, this investment yields more accurate performance predictions, which in turn improves the selection of final configurations. Appendix B.2 lists the hidden features, highlighting those with high importance in performance prediction.

4 Conclusion and Future Work

In this paper, we introduce ML^{2}Tuner, a multi-level machine learning tuning technique designed to enhance the efficiency and effectiveness of autotuning for DL accelerators. By incorporating a validity prediction model (Model V) to filter out invalid configurations and an advanced performance prediction model (Model A) that takes advantage of hidden features extracted during the compilation process, ML^{2}Tuner addresses key limitations of existing autotuning methods that rely on a single machine learning model.

As future work, we plan to evaluate ML^{2}Tuner on a diverse range of hardware platforms beyond VTA-style DL accelerators to assess its generalizability and effectiveness across various architectures. Additionally, we aim to incorporate advanced machine learning techniques, such as reinforcement learning and Bayesian optimization, to further enhance the tuning process. Furthermore, we intend to develop a self-recovering system capable of automatically handling runtime errors during tuning, thereby improving robustness and reducing the need for manual intervention. These efforts aim to enhance the adaptability, efficiency, and practicality of ML^{2}Tuner in optimizing DL models across various hardware environments.

Acknowledgment

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00459797, Development of ML compiler framework for on-device AI), (No.RS-2023-00277060, Development of open edge AI SoC hardware and software platform) and (No.2022-0-00454, Technology development of smart edge device SW development platform).

References

[1] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. Ai and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[2] Epoch AI. Key trends and figures in machine learning, 2023. Accessed: 2024-09-22.

[3] Ashutosh Mishra, Jaekwang Cha, Hyunbin Park, and Shiho Kim. *Artificial Intelligence and Hardware Accelerators*. Springer, 2023.

[4] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 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*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Cuda toolkit. `https://developer.nvidia.com/cuda-toolkit/`. Accessed: 2024-09-22.

[6] Intel Corporation. Intel gaudi 3 ai accelerator white paper. White Paper 817486, Intel, June 2024. Version 1.1.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015.

[8] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024.

[9] ONNX Runtime developers. ONNX Runtime, November 2018.

[10] OpenVINO™ Toolkit. Accessed: 2024-09-22.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[12] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[13] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[14] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[15] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *CGO 2021*, 2021.

[17] Nest compiler. `https://github.com/etri/nest-compiler`, 2021. Accessed: 2024-09-22.

[18] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

[19] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: adaptive tensor program compilation made efficient. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[20] Arya Fayyazi, Mehdi Kamal, and Massoud Pedram. Arco:adaptive multi-agent reinforcement learning-based hardware/software co-optimization compiler for improved performance in dnn accelerator design, 2024.

[21] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. Dynatune: Dynamic tensor program optimization in deep neural network compilation. In *International Conference on Learning Representations*, 2021.

[22] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[23] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[24] Xi Zeng, Tian Zhi, Zidong Du, Qi Guo, Ninghui Sun, and Yunji Chen. Alt: Optimizing tensor compilation in deep learning compilers with active learning. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 623–630, 2020.

[25] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 89–103, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[27] Jun Bi, Qi Guo, Xiaqing Li, Yongwei Zhao, Yuanbo Wen, Yuxuan Guo, Enshuai Zhou, Xing Hu, Zidong Du, Ling Li, Huaping Chen, and Tianshi Chen. Heron: Automatically constrained high-performance library generation for deep learning accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 314–328, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 530–543, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Shamik Kundu, Suvadeep Banerjee, Arnab Raha, Suriyaprakash Natarajan, and Kanad Basu. Diagnnose: Toward error localization in deep learning hardware-based on vta-tvm stack. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(1):217–229, 2024.

[30] Dennis Rieber, Moritz Reiber, Oliver Bringmann, and Holger Fröning. Hw-aware initialization of dnn auto-tuning to improve exploration time and robustness, 2022.

[31] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 530–543, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization, 2019.

[33] Zcu102 evaluation board user guide. `https://docs.amd.com/v/u/en-US/ug1182-zcu102-eval-bd`. Accessed: 2024-09-22.

[34] Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions, 2018.

[35] Vta library customized by ones ai. `https://gitlab.com/ones-ai/vta_lib`, 2024. Accessed: 2024-09-22.

[36] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[38] Supported xilinx target zcu104 with hardware preset. `https://github.com/apache/tvm-vta/blob/36a91576edf633479c78649e050f18dd2ddc8103/config/zcu104_sample.json`. Accessed: 2024-09-22.

[39] Vta configuration. `https://tvm.apache.org/docs/topic/vta/dev/config.html`, 2022. Accessed: 2024-09-22.

[40] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011.

[41] Chris J.C. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical Report MSR-TR-2010-82, June 2010.

[42] Chen Wang, Chengyuan Deng, and Suzhen Wang. Imbalance-xgboost: leveraging weighted and focal losses for binary label-imbalanced classification with xgboost. *Pattern Recognition Letters*, 136, 06 2020.

A Additional Experimental Information

A.1 Hardware

The extended VTA was implemented on the Xilinx ZCU FPGA, with the configuration details outlined in Table 1. FPGA boards, including the Intel DE10 Nano, Xilinx PNYQ, Xilinx ZCU104 [38], and Ultra96, are equipped with default VTA presets provided by TVM. However, no preset information was available for the Xilinx ZCU102 used in the experiments.

To proceed with the research on the Xilinx ZCU102, some adjustments were made to the VTA configuration based on [39]. First, the metadata of the VTA configurations, namely "Target" and "HW VER" were modified. The following attributes were left unchanged: (1) LOG INP WIDTH, (2) LOG WGT WIDTH, (3) LOG ACC WIDTH, (4) LOG BATCH, and (5) LOG BLOCK. The remaining four attributes were adjusted as follows: (1) LOG UOP BUFF SIZE, (2) LOG INP BUFF SIZE, (3) LOG WGT BUFF SIZE, and (4) LOG ACC BUFF SIZE, with each value increased by 1.

Table 1: Parameters of VTA Configurations

Attribute	Format	Description	Value
TARGET	String	The TVM device target.	zcu102
HW VER	String	VTA hardware version number.	0.0.1
LOG INP WIDTH	Int (log2)	Input data type signed integer width.	3
LOG WGT WIDTH	Int (log2)	Weight data type signed integer width.	3
LOG ACC WIDTH	Int (log2)	Accumulator data type signed integer width.	5
LOG BATCH	Int (log2)	VTA matrix multiply intrinsic input/output dimension 0.	0
LOG BLOCK	Int (log2)	VTA matrix multiply inner dimensions.	4
LOG UOP BUFF SIZE	Int (log2)	Micro-op on-chip buffer in Bytes.	16
LOG INP BUFF SIZE	Int (log2)	Input on-chip buffer in Bytes.	16
LOG WGT BUFF SIZE	Int (log2)	Weight on-chip buffer in Bytes.	19
LOG ACC BUFF SIZE	Int (log2)	Accumulator on-chip buffer in Bytes.	18

A.2 Neural Network

Table 2 provides an analysis of the convolution layers of ResNet18 and highlights the ratio of invalid configurations in VTA. Part (a) provides detailed information about the parameters of each convolution layer, including the input shape, kernel filter, output shape, padding, and stride for each layer. Part (b) presents the invalidity ratio of configurations for each convolution layer.

Table 2: Profiling Layer of Convolution in ResNet18 and Ratio of Validity Configurations on VTA

Name	(a) Information				(b) Invalidity Ratio
	H, W, C	KC, KH, KW	OH, OW	Pad, Stride	
Conv 1	56,56,64	64,3,3	56,56	1,1	0.8264
Conv 2	56,56,64	128,1,1	28,28	0,2	0.7966
Conv 3	56,56,64	128,3,3	28,28	1,2	0.8057
Conv 4	28,28,128	128,3,3	28,28	1,1	0.6935
Conv 5	28,28,128	256,1,1	14,14	0,2	0.5249
Conv 6	56,56,64	128,1,1	28,28	0,2	0.5249
Conv 7	56,56,64	128,3,3	28,28	1,2	0.5249
Conv 8	28,28,128	128,3,3	28,28	1,1	0.5047
Conv 9	56,56,64	128,3,3	28,28	1,2	0.5047
Conv 10	28,28,128	128,3,3	28,28	1,1	0.5047

The optimizable features in our VTA implementation and backend compiler are based on tiling and the number of virtual threads. An invalid configuration is defined as one in which a calculation fails due to a register error, requiring a manual reboot, or a test fails because the result differs from the expected result. Conversely, a valid calculation is a configuration that successfully completes the task without errors.

A.3 Experimental HyperParameter on ML^{2}Tuner

Models P and A are designed to identify the highest predicted performance, while Model V is configured for classification tasks. Consequently, Models P, A, and V have distinct objective functions and loss functions, which lead to different search ranges for these parameters during hyperparameter tuning. Models P and A were optimized using regression and ranking objectives to achieve the highest prediction performance. In contrast, Model V was optimized using binary classification objectives to define its search range.

To optimize the hyperparameters of XGBoost, we conducted a grid search [40] based on the search space outlined in Table 3. The overall tuning results are presented in Table 4.

Table 3: Exploration Range and Hyperparameters for XGBoost Models

Parameter	Search Space	Model P	Model V	Model A
objective	–	reg:squarederror	binary:hinge	reg:squarederror
boost round	–	300	300	300
max depth	$\{x \in \mathbb{N} \mid 3 \leq x \leq 15\}$	14	5	14
min child weight	$\{x \in \mathbb{R} \mid 1 \leq x < 10\}$	3	3	3
gamma	$\{x \in \mathbb{R} \mid 0.0 \leq x < 1.0\}$	0.0	0.0	0.0
subsample	$\{x \in \mathbb{R} \mid 0.4 \leq x \leq 1.0\}$	1.0	0.6	1.0
colsample bytree	$\{x \in \mathbb{R} \mid 0.5 \leq x \leq 1.0\}$	1.0	0.6	1.0
learning rate	$\{0.001, 0.01, 0.1, 0.2, 0.3\}$	0.01	0.1	0.01
reg alpha	$\{x \in \{-5, -2, -1, 0, 1\} \mid 1 \times 10^x\}$	1×10^{-5}	1×10^{-2}	1×10^{-5}

B Additional Experimental Results

B.1 Comparison of HyperParameter for Model V and Model A

The accuracy and training time were calculated by changing the objective function. The time and accuracy for the 10 ResNet18 layers in VTA are shown in Table 1.

Models P and A were configured using both Ranking and Regression objectives to compare their prediction performance, while Model V assessed the validity of configurations using both Binary classification and Regression objectives. For performance comparison, the hyperparameters are shown in Table 3.

As shown in Table 4, Model P and A achieved the highest prediction performance when using Regression objectives, outperforming Rank objectives by 0.06%p in accuracy and being 1.70x faster in computation time. In Model V, the difference in computation time required for different objective functions was greater than the difference in accuracy.

Table 4: Comparison of ML^{2}Tuner with objective function and loss

Model	Objective Function	Loss	Accuracy	Time (sec)
Model P and A	Regression	Squared Error	99.55	320.21
	Rank [41]	Logistic	99.49	537.74
Model V	Regression	Squared Error	99.49	316.23
		Logistic	99.47	350.89
	Binary [42]	Hinge	99.41	176.73
		Logistic	99.55	537.74

B.2 Impact of HyperParameters

The layer and kernel information was not displayed during the evaluation of the visible features. Table 5 provides an overview of both visible and hidden features values. The visible features are highlighted in blue, while the hidden features are presented in **black**. In addition, the impact of

each hyperparameter is expressed as a percentage, offering insight into their relative importance and contribution to the model's performance.

Three parameters significantly influence the visible features and have a substantial impact on the model's execution time: Tile Width(TW), Tile Height(TH), and Number of Virtual Threads(nVT). The TW and TH parameters are used in tiling optimization to divide operations into smaller blocks. The value of nVT represents the number of virtual threads involved in parallel operations.

Hidden features consist of information whose values are derived from visible features or collected through internal branching mechanisms. These features may include derived parameters or flags that influence execution paths. In cases where a feature name contains a Boolean operation, this means that the value of the feature is binary, either true or false, and represents the presence or absence of a particular condition or operation. If a Boolean operation is enclosed in parentheses within the feature name, it indicates that the variable's value depends on the specific branch taken during execution.

Table 5: Importance of Visible and Hidden Features

Feature	GeoAVG	Normalized Feature Importance Score (%)									
		Conv1	Conv2	Conv3	Conv4	Conv5	Conv6	Conv7	Conv8	Conv9	Conv10
TW	29.268	19.685	27.412	23.810	25.779	32.631	30.633	28.432	31.596	31.975	32.582
TH	25.925	15.256	19.768	20.186	22.288	27.235	29.335	29.201	32.406	31.028	30.622
nVirtualThread > 0 (threadIdx)	8.468	10.581	8.434	9.058	9.130	6.937	6.231	7.941	8.101	7.579	7.594
nVT	8.194	5.413	7.907	7.505	7.519	7.965	7.269	7.941	8.912	9.237	8.574
nFilterInLoop	4.933	7.382	5.271	5.435	4.565	3.854	4.154	3.842	4.321	4.737	3.920
sizeOutTileH	4.083	2.215	3.163	3.364	3.491	4.625	4.673	4.611	4.861	4.737	4.655
sizeOutTileW	4.166	2.707	4.217	3.364	3.491	5.139	4.413	4.098	4.051	4.500	4.655
nVirtualThread > 0 (threadIdx) 2	3.563	7.382	5.535	5.435	5.102	1.799	2.596	2.561	1.620	1.658	1.960
sizeOutTileBoundaryW	3.069	13.287	6.326	7.764	6.176	1.542	2.336	3.330	0.810	0.711	0.735
outDummyH(b0!=0)	1.946	0.738	1.581	1.812	2.685	2.569	2.596	2.561	1.080	1.184	1.715
nFilterInLoop	1.946	1.722	2.636	2.588	2.954	1.542	2.336	2.305	0.810	0.711	0.980
resizedOutTileH(b0==0)	1.233	9.104	3.426	3.623	1.611	0.257	0.260	0.512	0.270	0.237	0.245
outDummyH(b0==0)	1.069	1.476	1.845	3.623	2.148	0.771	0.779	1.025	0.270	0.237	0.245
Kn / nFilterInLoop / nVirtualThread / 16	0.740	0.492	0.527	0.518	0.806	1.799	1.038	0.768	0.270	0.474	0.490
sizeInTileW	0.411	0.246	0.264	0.259	0.269	0.514	0.519	0.256	0.270	0.474	0.490
resizedOutTileH(b0!=0)	0.274	0.738	0.791	0.776	1.343	0.257	0.260	0.256	0.027	0.024	0.024
sizeIntTileH	0.384	0.246	0.264	0.259	0.269	0.514	0.519	0.256	0.270	0.474	0.490
resizedInTileH(b0==0)	0.274	1.230	0.527	0.518	0.215	0.026	0.026	0.077	0.054	0.024	0.024
resizedInTileH(b0!=0)	0.055	0.098	0.105	0.104	0.161	0.026	0.026	0.026	0.001	0.000	0.000

B.3 Extended experimental results for all layers of ResNet18

Fig. 4 presents a comparison of the root mean squared error (RMSE) ratios between Model A and Model P across different layers, highlighting the impact of varying the number of boosting rounds and configuration samples. To compute the RMSE values displayed on the Y-axis, performance metrics were collected for all possible parameter configurations executable on the VTA. The dataset was split into training and test sets, with the training set generated by ML^{2}Tuner based on the specified number of configuration samples. To reduce experimental error, each experiment was repeated 10 times, and the average results were calculated.

The results for Models P and A were plotted as functions of the number of configuration samples and the number of boosting rounds in XGBoost. In these graphs, the Y-axis represents the ratio of RMSE values between $\text{RMSE}_{\text{Model P}}$ and $\text{RMSE}_{\text{Model A}}$, while the X-axis indicates the number of configuration samples.

The results indicate that, for most layers, Model A achieves higher accuracy on the test set after training compared to Model P. Notably, increasing the number of boosting rounds from 100 to 300 improves test set accuracy. Specifically, the average test set accuracy increases from 0.916 with 100 boosting rounds to 0.932 with 300 boosting rounds.

Fig. 5 illustrates the results of ResNet's convolution layers, emphasizing the effects of differences between validity prediction and advanced performance prediction. Layers 1 through 10 show that fewer invalid configurations are explored compared to the TVM method. Notably, ML^{2}Tuner achieves higher performance with fewer profiling attempts, particularly in Conv1 through Conv3, where the invalidity ratio is higher, thus outperforming TVM in terms of efficiency.

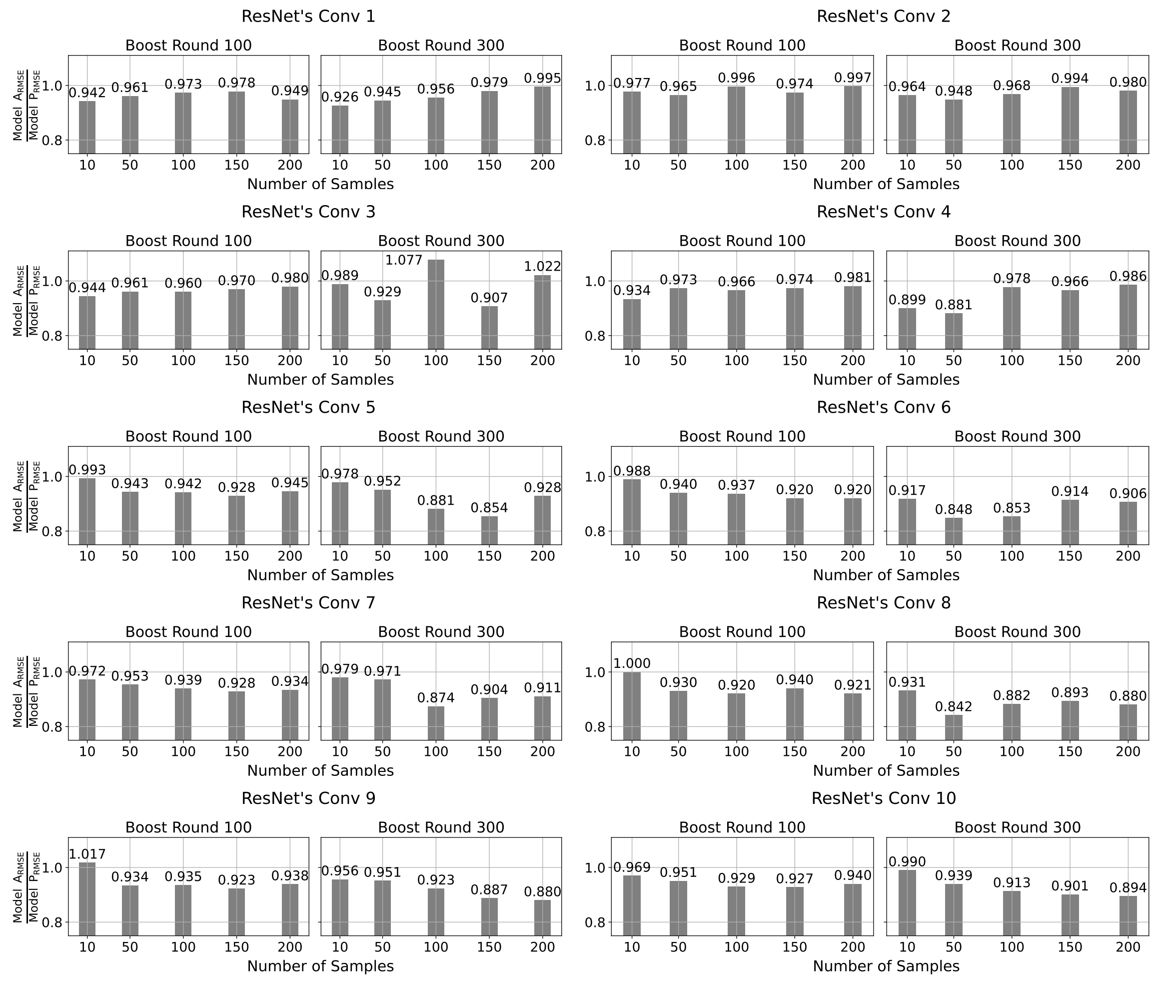

Figure 4: Ratio of RMSE values of model A compared to model P per layers

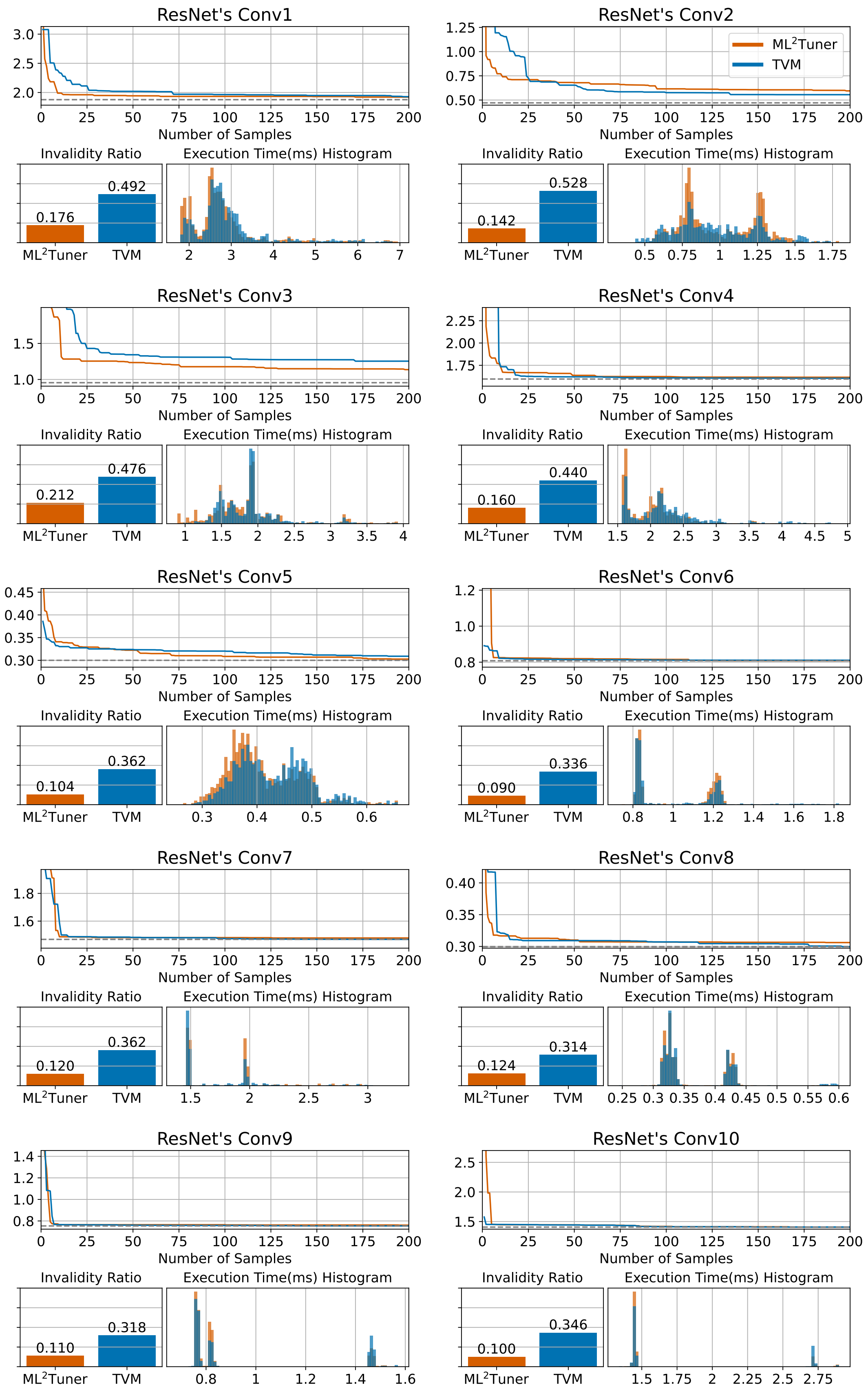

Figure 5: Result of ResNet18 per layer on ML^{2}Tuner