Programming Systems for Improving the Performance of Deep Learning-Intensive Applications

Yifan Zhao
yifanz16@illinois.edu
University of Illinois
Urbana-Champaign, USA

Vikram Adve
vadve@illinois.edu
University of Illinois
Urbana-Champaign, USA

Sasa Misailovic
misailo@illinois.edu
University of Illinois
Urbana-Champaign, USA

Abstract

Computationally intensive Neural Networks (NNs) are ubiquitous in applications such as natural language processing, autonomous driving, and augmented reality. These NN applications are increasingly deployed in resource-constrained edge computing environments, making it challenging to perform inference at the application's performance requirement. Optimization of NN applications is the key to their efficiency, usability, and accessibility. On the other hand, optimization of NN applications is difficult, because diverse NN architectures, application designs, and target hardware require many optimization decisions to be made by compilers, or domain experts when current compilers do not suffice.

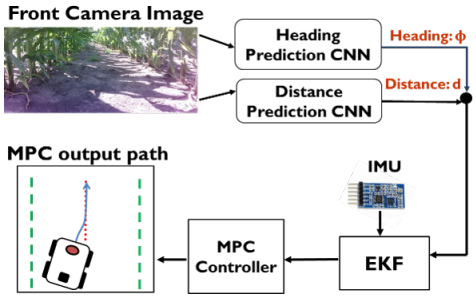
In this paper, we observe that two important opportunities for the optimization of NN applications exist: *end-to-end quality-aware optimization*, and *deep learning domain-specific compiler optimizations*. We assert that a system that is effective at optimizing NN applications need to combine both approaches, and leverage domain-specific and application-specific information. We demonstrate the effectiveness of these two approaches with two examples of NN application optimizers: *ApproxCaliper* (MLSYS'23) and *Felix* (ASPLOS'24), and list current challenges in both approaches that need to be addressed to create better NN optimizing systems.

ACM Reference Format:
Yifan Zhao, Vikram Adve, and Sasa Misailovic. 2024. Programming Systems for Improving the Performance of Deep Learning-Intensive Applications. In *Proceedings of 3rd Workshop on Practical Adoption Challenges of ML for Systems (PACMI'24).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Many emerging edge applications that make decisions combine deep learning components with non-DL components. Notably, autonomous systems combine multiple neural network (NN) models that extract actionable information from sensor data such as images and audio, other computations that process other sensory information such as LIDAR and GPS, and decision-making code (written in a conventional programming language) to achieve end-to-end goals. However, DL components are often compute- and power-intensive, which makes it challenging to deploy these models on resource-constrained edge compute devices with tight constraints on power, weight, size and production costs [12, 16].

(a) TerraSentia navigating in fields of crops.



(b) CropFollow navigation system workflow.

Optimizing such an application opens up two important opportunities that critically leverage *domain-specific information* and automated programming systems:

- **End-to-end quality-aware optimization:** Many applications exhibit a significant amount of *application-level error resilience*: reducing an individual component's accuracy is acceptable if it has minimal *observable impact on the end-to-end quality of the application.* Identifying how much component accuracy can be dropped while maintaining the overall quality and reliability of the system can be done with specialized accuracy-aware autotuners.
- **Leveraging specialized compilation strategies for components:** NNs consist of NN operators (*kernels*) such as convolution and matrix multiplication. Due to the diversity of NN operators and hardware platforms, obtaining high-performance kernels for all their combinations is a challenging task. While the current frameworks like PyTorch and TensorFlow provide kernel libraries, a more promising approach is to *automatically generate* kernels for each new target platform using ML compilers.

Example. As an illustrative example, we will use *TerraSentia*, a state-of-the-art commercial agriculture robot obtained from EarthSense [7]. It is used by farmers for high-throughput phenotyping and a variety of other agriculture tasks. The

robot is equipped with an autonomous vision-guided navigation system named *CropFollow* [22] used for *row-following* navigation through fields of crops (Figure 1a). CropFollow contains a number of components that collaborate to keep the robot in the center of a crop row, shown in Figure 1b. It includes 2 CNNs that take 320×240 RGB images as input and estimate the robot's current position as two quantities: driving angle and distance from the crop row edge. Multiple non-NN components process the output of these CNNs and guide the navigation. The CNNs are the main performance bottleneck of CropFollow and significantly raise its power consumption and the cost of hardware.

Quality-Aware Optimization. The prediction accuracy (or error) of NNs in an NN-based application is characterized by comparing the NN's output and the groundtruth. Meanwhile, the *quality of service (QoS)* of the application is concerned with the *correctness of behavior* of the entire application, which is a different objective from the accuracy of its NNs. For example, CropFollow's heading and distance prediction CNNs each output a scalar, and their prediction error is the l_1 distance between the prediction and the groundtruth. The QoS of CropFollow is its navigation quality, measured by the number collision with crops (0 expected) in a run of given distance. In CropFollow, a reduction in the accuracy of the CNNs may not adversely affect its QoS due to the error resilience in the control components. Therefore, CropFollow exhibits application-level error resilience. Application error resilience manifests in many different domains that use NNs, including autonomous navigation systems, augmented and virtual reality (AR/VR) stacks, and real-time data analytics.

Currently, application developers can reduce the costs of NNs with various NN-specific optimization techniques such as pruning, quantization, and low-rank factorization [3, 9–11, 18–21, 23, 24, 26]. These techniques optimize a neural network with the goal to *maintain the same accuracy as the original model.* They have proven effective in many scenarios where the network *is* the entire application, but on composite edge applications, these techniques miss the rich opportunity to take advantage of application-level resilience.

Incorporating this application-level resilience in the optimization process allows relaxing the accuracy of NN components and applying more aggressive optimizations. However, tuning the approximation settings for NN components is complicated and time-consuming. because the search space is often large, and every application QoS evaluation is expensive. CropFollow has two NN components where errors in one affect how much error can be tolerated from the other. Jointly tuning the approximation settings of these two NNs results in a search space of a few thousand configurations. One evaluation of the QoS of CropFollow requires a 100m run in real crop fields, which takes more than 5 minutes.

Our experiments (§2.1) show that, a system that can overcome these challenges can yield significant performance improvements. *Application-aware optimizations* can provide several times more speedup and over an order of magnitude less memory consumption compared to application-agnostic techniques. These improvements make it possible to deploy compute-intensive NN models on edge systems.

Domain-Specific Compiler Optimizations. Another aspect of optimizing NN-based applications is to *use high-performance implementations (kernels)* for NN operators such as convolution and matrix multiplication. Due to the diversity of NN operators and hardware platforms, obtaining high-performance kernels for all their combinations is a challenging task. Existing DL frameworks, such as PyTorch [15] and TensorFlow [1], map operators in NNs to *kernel libraries* with manually optimized kernels for specific hardware architectures, such as cuDNN for NVIDIA GPUs and MKL-DNN for Intel CPUs. However, this approach requires significant expertise and manual effort, and the optimized code does not carry across the increasingly diverse edge platforms.

Automatic code generation of NN kernels [2, 4–6, 8, 17] is a recent and improved approach to finding efficient implementations of NN operators. To handle the diversity of hardware platforms, these *tensor compilers* use a search-based code generation approach. A tensor compiler typically defines a search space of *schedules*: sequences of program transformations such as loop tiling, vectorization, and parallelization, to apply to the user-given initial program. The code generator then searches for a schedule that delivers high performance for this program on the target hardware.

To achieve good performance for a wide range of different hardware architectures, the search space needs to include a large number of candidate schedules. However, searching in a large space is fundamentally difficult. Existing tensor compilers rely on combinatorial discrete search techniques such as beam search and/or genetic algorithms, and suffer from excessively long tuning time of hours or days per program [17, 29]. Some existing approaches therefore resort to covering only part of the search space, using manually-written templates [5, 6] or aggressive pruning [2], thereby limiting the performance benefits.

Bringing It Together. Together, these two directions, approach the full application optimization from two opposite directions: outside-in from the end-to-end quality/latency requirements to component-level requirements (quality-aware optimization) and inside-out from kernel-level optimizations to full application optimization (ML component optimization). Our vision for fully optimizing modern edge applications with ML components:

> Domain-specific and application-specific knowledge is critical for the optimization of DNN applications; we can better design programming systems that leverage this knowledge with minimal user effort, and optimizes DNN applications from both directions to maximize the benefits.

2 Toward End-to-End Quality-Aware Optimization

2.1 ApproxCaliper

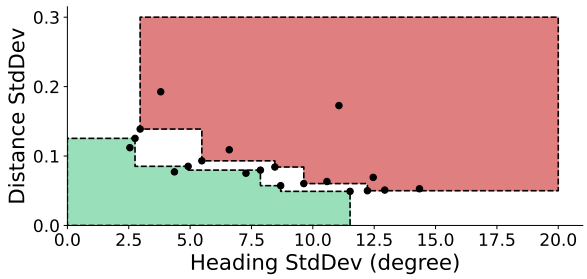


Figure 2. Error constraint space ApproxCaliper extracted from CropFollow. It captures the interaction of errors in the heading CNN (x-axis) and the distance CNN (y-axis).

ApproxCaliper [28] is the first application-aware neural network optimization framework. It is application-aware in that it uses a developer-specified application-level QoS goal for tuning. For instance, when optimizing the CropFollow system, a developer can specify a QoS goal that TerraSentia should autonomously navigate without collisions for a given distance. ApproxCaliper encodes this QoS goal as constraints under which it relaxes NN accuracy with approximation techniques to gain higher performance benefits.

To reduce the complexity and cost of QoS-performance autotuning, ApproxCaliper's novel optimization algorithm starts from the observation that the application QoS of a configuration depends on the error levels of all NN components. It thus reduces the problem of searching the application QoS space to searching the local spaces of NN errors measured by NN-specific error metrics, which are much cheaper to evaluate.

ApproxCaliper presents a two-phase optimization approach with an error calibration phase and a model tuning phase. Its novel **error calibration algorithm** uses statistical error injection to identify valid regions in the NN error space (i.e., regions of configurations that lead to acceptable application-level QoS) and separate them from the invalid regions. The result of this phase is an *error constraint space*, which guides the following **model selection and tuning phase** towards configurations that maximize the given objective while satisfying the error constraints.

Figure 2 shows the error constraint space that ApproxCaliper's first phase extracted from CropFollow. The NN error metrics of the 2 NNs in CropFollow are placed on the x and y axes, while the color indicates if the NN errors at a point lead to valid application QoS level or not. ApproxCaliper charts this space in only 20 empirical QoS evaluation (running TerraSentia in the field), and these evaluations are shown as black dots. Using this error constraint space, ApproxCaliper optimizes CropFollow and provides 5.8× *more* speedup than application-agnostic approximation – the practice of applying NN approximations while preserving NN accuracy – while not affecting CropFollow navigation QoS. As a result of this performance improvement, we were able to reduce the compute hardware on TerraSentia to a Raspberry Pi4 ($35) with all CNN inferences running on its CPU.

2.2 Felix

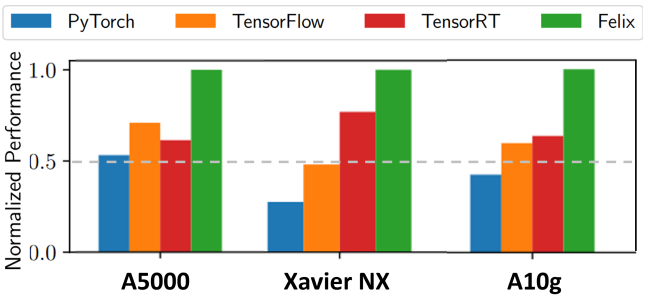


Figure 3. DNN inference performances using Felix vs. deep learning frameworks (PyTorch, TensorFlow, TensorRT) on three HW platforms.

Felix [27] is a novel *gradient-based* compiler optimization framework for tensor programs that greatly improves the search efficiency in automatic kernel generation. Felix deviates from the common practice of relying on combinatorial search algorithms for schedule tuning, and instead applies gradient descent over a *differentiable performance prediction function* to optimize tensor programs. Felix applies continuous relaxation on the space of programs and creates differentiable estimator of program latency, allowing efficient search of program candidates using gradient descent, in contrast to conventional approaches that search over a non-differentiable objective function in a discrete search space.

To enable gradient in the program schedule search and formulate the program schedule search as a differentiable optimization problem, Felix overcomes the following significant challenges:

- The search space of schedules is discrete, with many of the tunable parameters constrained in a subset of integers. For example, the tiling sizes of loop tiling optimization must be integers, but also must be factors of the loop extent to not introduce conditional branches.
- The *objective function* – the performance of the program as a function of the schedule – is highly complex, often discontinuous, and non-differentiable. To evaluate a schedule accurately, the compiler needs to generate a program from the schedule and empirically measure the performance of the program on the target hardware.
- Analytical [25] or learned [13, 14, 30] performance models approximate the performance objective and are generally

faster to evaluate than empirical measurements. However, to maintain generality, these models typically take as input the generated program or a program *feature vector* representative of the program's performance, instead of taking the schedule as the input. Even when these models are differentiable themselves, differentiation of the feature vector with regard to the variables in the program schedule remains tremendously challenging as program generation exercises multiple components of the compiler.

We evaluate Felix on 6 DNNs and 3 GPU hardware platforms. As Figure 3 shows, Felix improves the performance of the 6 networks on average (geometric mean) by 1.41× on A5000, 1.50× on A10G and 1.70× on Xavier NX, compared to off-the-shelf inference frameworks PyTorch, Tensorflow, and TensorRT (up to 4.48×, 5.40×, and 10.8× respectively). Additional experiments show that Felix surpasses the performance of these frameworks within only 7 minutes of search time on average. Compared to TVM Ansor, which autotunes NN kernels using discrete search method (genetic algorithm), Felix's search converges much faster, reaching a 95% performance of the best discovered code 3.4× faster on average (geomean). Felix is particularly effective for time-constrained tuning or tuning on resource-constrained edge devices, as Felix can quickly find schedules with high performance.

3 Challenges for End-to-End Quality-Aware Optimization

The success of ApproxCaliper and Felix demonstrates the greater potential of *full-application and multi-objective optimization* for NN applications. We envision the following improvements over the current state of the art in order to fully realize these potentials:

Application-aware Optimizations. To achieve full-application quality-aware optimization, multiple NNs in the application need to be *co-optimized*, forming a larger search space where each point is a combination of all the NNs' approximation settings. In ApproxCaliper, to make the combined search space tractable, the size of each NN's search space is kept small. ApproxCaliper utilizes off-the-shelf NN approximation techniques that applies the approximation uniformly on each layer of the NN, and controls one parameter (e.g. the global pruning ratio) per NN. However, many approximation techniques are finer-grained and allows control over, for example, the pruning ratio per *layer* in the NN. While these techniques may provide better performance-accuracy tradeoff, they impose much larger search space per NN, and an intractable combined space when multiple NNs exist.

To enable these finer-grained approximations in quality-aware optimization, we need to both prune the search space and improve the search algorithm. For example, some approximation techniques employ inexpensive *sensitivity analyses* to analytically derive most approximation parameters, and only a few need to be searched or user-given. In addition, some techniques formulate approximation decision-making into a differentiable search problem and uses gradient descent to optimize it during the training of the NN. A system that extends the quality-aware optimization to support these techniques can provide even more effective optimizations.

Extended Full-DNN Optimizations. Graph-level optimizations, such as operator fusion, transcends the individual operator boundaries and offers additional optimization opportunities. When optimizing a DNN, existing tensor compilers typically apply a fixed set of graph-level optimizations, before breaking down the DNN into layers (or small groups of layers) and tune the schedule of each layer in isolation. The advantage of this approach is reducing the size of the search space, by viewing each tuning "task" as independent. However, it misses out on graph-level transformations that are not always profitable and need to be autotuned. For instance, fusing operators that both have reduction operations can lead to recomputation, where the profitablity highly depends on the size of the operators and the hardware.

NN approximations also typically operate at graph-level to change the input/output shapes or scalar types of multiple layers. A system that jointly optimizes accuracy and performance using both approximation and search-based kernel generation can make approximation decisions with *real kernel performance* instead of performance proxies such as FLOPs, as typical in today's practice. Because these approximation decisions influence the scheduling of individual kernel, the result is a combined search space that is immense in size. Efficient search techniques such as gradient descent are vital for exploring such global search spaces. However, graph-level optimizations also present a new set of challenges to Felix-style kernel schedule search, as graph transformations create more profound changes in the program structure that can be difficult to capture in one continuous space.

Acknowledgements

We thank the anonymous reviewers for their comments. This research was supported in part by the National Science Foundation (Grant No. CCF-2217144) and the IBM-Illinois Discovery Accelerator Institute.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38, 2019.

[3] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3), 2017.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman P. Amarasinghe. A deep learning based cost model for automatic code optimization. *CoRR*, abs/2104.04955, 2021.

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

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[7] Earthsense. A Growing Presence on the Farm: Robots. https://www.nytimes.com/2020/02/13/science/farm-agriculture-robots.html, 2020.

[8] Adel Ejjeh, Aaron Councilman, Akash Kothari, Maria Kotsifakou, Leon Medvinsky, Abdul Rafae Noor, Hashim Sharif, Yifan Zhao, Sarita Adve, Sasa Misailovic, and Vikram Adve. Hpvm: Hardware-agnostic programming for heterogeneous parallel systems. *IEEE Micro*, 42(5):108–117, 2022.

[9] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[10] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[11] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[12] Jong-Hoon Kim, Gokarna Sharma, and S Sitharama Iyengar. Famper: A fully autonomous mobile robot for pipeline exploration. In *2010 IEEE International Conference on Industrial Technology*. IEEE, 2010.

[13] Paolo Sylos Labini, Marco Cianfriglia, Damiano Perri, Osvaldo Gervasi, Grigori Fursin, Anton Lokhmotov, Cedric Nugteren, Bruno Carpentieri, Fabiana Zollo, and Flavio Vella. On the anatomy of predictive models for accelerating gpu convolution kernels and beyond. *ACM Trans. Archit. Code Optim.*, 18(1), jan 2021.

[14] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. *Advances in Neural Information Processing Systems*, 32, 2019.

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

[16] Søren Marcus Pedersen, Spyros Fountas, Henrik Have, and BS Blackmore. Agricultural robots—system analysis and economic feasibility. *Precision agriculture*, 7(4), 2006.

[17] 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. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[18] Xiaofeng Ruan, Yufan Liu, Chunfeng Yuan, Bing Li, Weiming Hu, Yangxi Li, and Stephen Maybank. Edp: An efficient decomposition and pruning scheme for convolutional neural network compression. *IEEE Transactions on Neural Networks and Learning Systems*, 32(10), 2021.

[19] Tara N Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013.

[20] Hashim Sharif, Prakalp Srivastava, Muhammad Huzaifa, Maria Kotsifakou, Keyur Joshi, Yasmin Sarita, Nathan Zhao, Vikram S Adve, Sasa Misailovic, and Sarita V Adve. Approxhpvm: a portable compiler ir for accuracy-aware optimizations. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[21] Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Akash Kothari, Ben Schreiber, Elizabeth Wang, Yasmin Sarita, Nathan Zhao, Keyur Joshi, Vikram S Adve, Sasa Misailovic, and Sarita V Adve. Approxtuner: a compiler and runtime system for adaptive approximations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[22] Arun Narenthiran Sivakumar, Sahil Modi, Mateus Valverde Gasparino, Che Ellis, Andres Eduardo Baquero Velasquez, Girish Chowdhary, and Saurabh Gupta. Learned visual navigation for under-canopy agricultural robots. *CoRR*, abs/2107.02792, 2021.

[23] Sridhar Swaminathan, Deepak Garg, Rajkumar Kannan, and Frederic Andres. Sparse low rank factorization for deep neural network compression. *Neurocomputing*, 398, 2020.

[24] Swagath Venkataramani, Jungwook Choi, Vijayalakshmi Srinivasan, Wei Wang, Jintao Zhang, Marcel Schaal, Mauricio J. Serrano, Kazuaki Ishizaki, Hiroshi Inoue, Eri Ogawa, Moriyoshi Ohara, Leland Chang, and K. Gopalakrishnan. Deeptools: Compiler and execution runtime extensions for rapid ai accelerator. *IEEE Micro*, 39:102–111, 2019.

[25] Yao Wang, Xingyu Zhou, Yanming Wang, Rui Li, Yong Wu, and Vin Sharma. Tuna: A static analysis approach to optimizing deep neural networks. *CoRR*, abs/2104.14641, 2021.

[26] Ran Xu, Rakesh Kumar, Pengcheng Wang, Peter Bai, Ganga Meghanath, Somali Chaterji, Subrata Mitra, and Saurabh Bagchi. ApproxNet: Content and Contention-Aware Video Object Classification System for Embedded Clients. *ACM Transactions on Sensor Networks*, pages 11:1–11:27, 2021.

[27] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. Felix: Optimizing tensor programs with gradient descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 367–381, 2024.

[28] Yifan Zhao, Hashim Sharif, Peter Pao-Huang, Vatsin Shah, Arun Narenthiran Sivakumar, Mateus Valverde Gasparino, Abdulrahman Mahmoud, Nathan Zhao, Sarita Adve, Girish Chowdhary, Sasa Misailovic, and Vikram Adve. Approxcaliper: A programmable framework for application-aware neural network optimization. *Proceedings of Machine Learning and Systems (MLSYS)*, 5, 2023.

[29] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020.

[30] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *NeurIPS; Datasets and Benchmarks Track*, 2021.