ACLTuner: A Profiling-Driven Fast Tuning to Optimized Deep Learning Inference

Yongin Kwon[1] **Joo Hyoung Cha**[2] **Jubin Lee**[3] **Misun Yu**[1] **Jeman Park**[1]
Jemin Lee[1*]
[1]Electronics and Telecommunications Research Institute [2]Dong-eui University [3]Neubility
{yongin.kwon,msyu,jeman,leejaymin}@etri.re.kr
{joohyoung.cha,jubin0927}@gmail.com

Abstract

Deep learning has expanded its footprint across diverse domains. The performance of these computations hinges on the interplay between deep learning compilers and inference libraries. While compilers adapt efficiently to new deep learning operations or models, their tuning processes are too time-consuming. In contrast, inference libraries offer quick execution but with adaptability limitations. To address these challenges, we propose ACLTuner, which optimizes execution configurations using existing inference library kernels. ACLTuner identifies and assigns the optimal kernel through targeted device profiling. Compared to ArmNN, AutoTVM, Ansor, ONNXRuntime, and TFLite, ACLTuner not only achieves up to 2.0x faster execution time across seven deep learning models, but also reduces the average tuning time by 95%.

1 Introduction

Deep learning has exhibited remarkable precision across tasks such as image/video analysis, audio/speech recognition, and natural language processing. Consequently, its integration has increased in diverse areas like autonomous systems, art, and manufacturing. With continuous advancements in GPU technology, the speed and efficiency of deep learning computations have seen significant improvements, enhancing learning and inference techniques. However, in scenarios limited by costs, energy, and physical space, a persistent push remains to harness conventional CPUs or innovate new hardware solutions for deep learning applications.

During inference, the efficiency of deep learning execution is mainly influenced by deep learning compilers [1, 2, 3, 4] or inference libraries [5, 6, 7, 8, 9]. Given adequate computational resources and time, deep learning compilers generate execution code that is optimized for the specific target hardware. Notably, AutoTVM [10] and Ansor [11] create a diverse range of code configurations, which are then profiled on actual devices. These compilers are adept at selecting optimal code versions to fully exploit the hardware's capabilities.

On the other hand, the moment a deep learning model is provided, inference libraries initiate computational processes promptly on the target device. These libraries include highly optimized kernels that handle the computation of the smallest units of deep learning operations. During execution, based on predetermined rules, the libraries select appropriate kernels and allocate them to the target hardware to process the entire deep learning operations. As a result, these libraries deliver outstanding performance for standard models without the need for pre-compilation.

To optimally execute unique and novel deep learning operations, compilers are advantageous. However, searching exhaustively through all code configurations is exceedingly time-intensive. Machine learning based auto-tuning algorithms [10, 11, 12, 13, 14, 15, 16] are employed to mitigate this time,

*Corresponding author.

Machine Learning for Systems Workshops at 37th NeurIPS 2023.

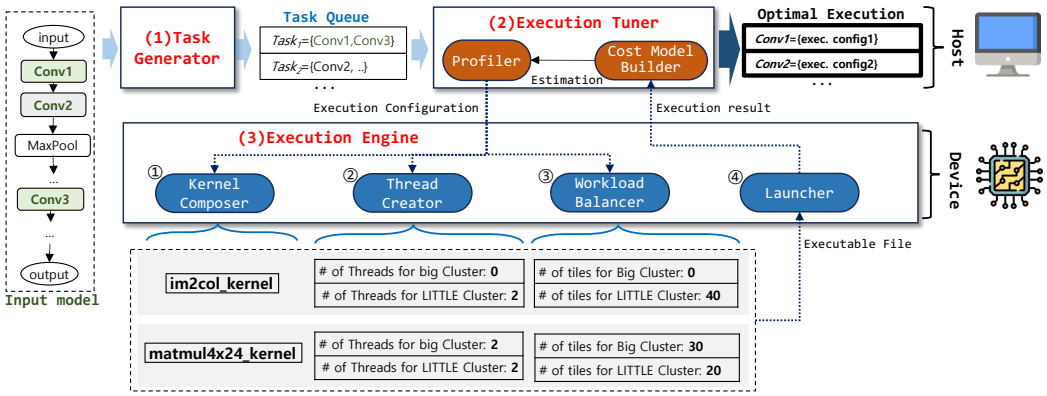

Figure 1: Overall tuning process of ACLTunner

but it still often takes many hours. Once an optimal execution is identified, it can be incorporated into the inference library as a kernel for reuse. Yet, it is challenging to individually implement kernels and define rules for every other operation that is not popular. Hardware that shares an Instruction Set Architecture(ISA), such as AMD64 and AArch64, has the capability to execute the same code. However, this does not guarantee that an optimal execution for one hardware will be equally optimal for another. As a result, it is crucial to identify and tailor the best execution for each specific hardware configuration.

In this paper, we introduce ACLTuner, an automated methodology that produces optimized executions by utilizing the existing kernels found in inference libraries. By conducting profiling on the target devices(Edge R and Odroid N2+), ACLTuner identifies the best-suited kernel and its optimal allocation. To demonstrate the effectiveness of our proposed method, we conducted experiments comparing ACLTuner with ArmNN [9], AutoTVM [10], Ansor [17], ONNXRuntime [5], and TFLite [18]. The experimental results showed that ACLTuner achieved a speedup of up to 2.0x in the execution time for seven representative deep learning models. Moreover, compared to AutoTVM and Ansor, ACLTuner reduced the average tuning time by 95%.

2 ACLTuner: Optimizing Compute Libraries for Diverse Hardware

2.1 Performance Challenges on Heterogeneous Architectures

Inference libraries feature extensively pre-optimized kernels. The difficulty arises in selecting and allocating them appropriately, based on the specific deep learning operation and the target hardware. In the context of ARM's big.LITTLE CPU architectures [19], various implementations are possible depending on the processor type and configuration. In particular, in these architectures, identical code execution can yield significantly different performance results.

Table 1: The execution time of ResNet18's Conv5 layer with different execution configurations on Edge R and Odroid N2+. The default configuration's kernel kind, number of created threads, and workload distribution ratio are determined by the rule of ArmNN.

Selected kernel	# of Threads & Workload ratio	Execution time(ms) Edge R	Odroid N2+
(**Default**)Winograd	1:1	7.8	13.0
Winograd	1:1:1:1:3:3	**6.1**	11.4
Direct Convolution	1:1:2:2:2:2	11.1	**4.4**

To validate our hypothesis, we conducted tests to assess the performance of deep learning operations on two edge devices, Edge R[20] and Odroid N2+[21], both equipped with ARM's big.LITTLE CPUs. When executing the fifth convolution layer of ResNet18 on each device using ARM's inference library, the library's rule consistently created two threads on both devices to run the Winograd kernel and distributed the workload evenly between them. *Workload* refers to the number of kernel invocations for a deep learning operation. As listed in Table 1, the experimental results show that the execution times on Edge R and Odroid N2+ were 7.8 and 13.0 ms, respectively. When we vary the number of threads and the workload distribution on Edge R, the execution time is improved to 6.1 ms. Interestingly, on Odroid N2+, performance increases by almost 200% when using direct convolution instead of Winograd. However, this change leads to performance degradation on the Edge R device.

2.2 ACLTuner System Design

ACLTuner is designed to automatically identify the optimal kernel and workload distribution with minimal profiling on real devices. To achieve this objective, ACLTuner consists of *Task Generator*, *Execution Tuner*, and *Execution Engine*.

Task Generator first generates tasks in a task queue that require tuning from the input model, as shown in Figure 1. During this process, operations that have identical input, output, and weight dimensions are grouped together to form a single tuning task, thus reducing the tuning time. In the case of ResNet101, for example, 104 operations require tuning, but only 23 tasks are generated. The generated tasks are placed in the task queue so that the following steps are performed for each task.

Execution Tuner searches for the optimal execution configuration for each task in the task queue by executing them directly on the device. To find the optimal execution configuration, the search space is calculated as *# of convolution algorithms* $\times$ *# of kernel types* $\times$ *# of workload distribution cases*($\approx 10^{12}$). Given the vast search space, the challenge of thoroughly exploring all possible execution configurations is daunting. Therefore, a *cost model builder* trains a *cost model* for each task to identify the optimal execution configuration with the fewest samplings. A *profiler* utilizes the *cost model* to efficiently determine the optimal execution configuration for each operation. The details of the process are as follows. (1) It randomly creates an execution configuration and initiates training of the *cost model*. (2) After initial training, we measure execution time by combining the configurations recommended by the *cost model* and those randomly generated. The *cost model* is then re-trained with new data. If we only use the execution configurations recommended by the *cost model* for training, it may lead to overfitting the training data. Thus, we exploit random data to mitigate this issue. (3) Using the newly trained *cost model*, we repeat the process in step (2), searching for an optimal execution configuration. Ultimately, an optimal execution configuration is saved. (4) From the *task queue*, we retrieve the next operation and repeat the entire process from step (1) to step (3).

Execution Engine sequentially executes a *kernel composer*, a *thread creator*, and a *workload balancer* to convert the execution configuration received from the *execution tuner* into an executable file. The *kernel composer* combines one or more kernels from the existing library to tailor them for the target operation. Direct convolution algorithm, for example, can be configured with a single matrix multiplication(MM) kernel. In contrast, Winograd [22] algorithm requires pre- and post-processing kernels in addition to the MM kernel. Figure 1 shows an example of being composed of two distinct `kernels:im2col` and `matmul4x24`. The *thread creator* creates dedicated threads for each CPU cluster. Depending on the execution configuration, threads may not be created for a specific cluster. Based on the execution configuration, the *workload balancer* allocates the workload to each thread. Once an executable file is created in this manner, a *launcher* reads it to execute and then delivers the output and execution time to the *execution tuner* for training the *cost model*.

Once the tuning is finalized, the optimal execution configuration is produced and can be run without any additional tuning next time.

3 Experiments

3.1 Implementation and Experimental Setup

The first version of ACLTuner is implemented based on ArmNN. We verified that ACLTuner is not limited to implementation in ARM's library; it can also be integrated with XNNPack [23], which is employed by TFLite, Pytorch Mobile [24], Alibaba HALO [25], and Samsung ONE [26], or directly within ONNXRuntime's native BLAS library. We employed XGBoost [27] to train regression models that serve as the cost model.

We performed experiments on representative deep learning models, including ResNet18[28], ResNet50, ResNet101, AlexNet[29], VGG16[30], GoogLeNet[31], and MobileNetV2[32]. To demonstrate that ACLTuner works well even with atypical models, we implemented ResNetXY. This model is composed of layers with kernel numbers and sizes randomly selected from ResNet50, All models are trained with ImageNet[33] dataset. The experiments are carried out on two edge devices, Tinker Edge R[20] (4-core A53 and 2-core A72) and Odroid N2+[21] (2-core A53 and 4-core A73).

ACLTuner was tested against ArmNN(v22.05), AutoTVM(v0.10), Ansor(v0.10), ONNXRuntime(v1.15.1), and TFLite(v2.14). AutoTVM and Ansor have tuning processes similar to ACLTuner, finding optimal execution codes in advance, while the others execute inference directly, without any tuning process. For TFLite, it is necessary to pre-define the number of threads that will be used. We

Table 2: End-to-end performance comparison on Edge R and Odroid N2+. The baseline is the inference time (ms) with ArmNN. We compared the results of ACLTuner with AutoTVM (TVM), Ansor, ONNXRuntime (OR), and TFLite (TF). Compared to the baseline, the speedup is represented as a factor (x). For ACLTuner, AutoTVM, and Ansor, which require tuning, the tuning time is denoted in hours (h) and is provided in parentheses.

	Edge R						Odroid N2+					
Model	ArmNN	**ACLTuner**	TVM	Ansor	OR	TF	ArmNN	**ACLTuner**	TVM	Ansor	OR	TF
ResNet18	195.66	**1.4x(1.6h)**	1.3x(27.1h)	0.8x(29.2h)	0.9x	1.0x	138.64	1.7x(**1.6h**)	**1.9x**(10.1h)	1.4x(26.0h)	0.9x	1.3x
ResNet50	480.23	**1.6x(2.0h)**	1.2x(19.5h)	1.1x(27.1h)	1.0x	1.1x	389.63	**2.0x(1.9h)**	1.8x(6.2h)	1.5x(27.1h)	1.1x	1.7x
ResNet101	873.11	**1.6x(2.7h)**	1.1x(51.5h)	1.0x(46.2h)	0.8x	1.1x	651.63	**2.0x(2.6h)**	1.7x(23.2h)	1.4x(34.7h)	0.9x	1.6x
AlexNet	124.36	**1.6x(0.4h)**	0.8x(33.4h)	0.6x(32.7h)	1.0x	1.2x	78.47	1.3x(**0.4h**)	0.6x(53.7h)	**1.4x**(25.8h)	0.8x	1.1x
VGG16	915.24	**1.5x(0.9h)**	1.0x(33.3h)	0.5x(30.4h)	0.6x	0.6x	701.27	**1.6x(0.8h)**	1.2x(31.0h)	0.8x(25.8h)	0.7x	0.9x
GoogLeNet	210.26	**1.4x(2.4h)**	1.3x(49.8h)	1.1x(28.5h)	1.2x	1.1x	147.21	1.6x(**2.3h**)	**1.6x**(22.6h)	1.5x(23.2h)	1.2x	1.6x
MobileNetV2	118.02	1.1x(**1.6h**)	1.7x(43.2h)	**1.8x**(62.7h)	1.5x	1.2x	83.02	1.5x(**1.5h**)	1.8x(49.8h)	**2.8x**(23.8h)	1.3x	1.4x
ResNetXY	1782.50	**1.8x(2.4h)**	0.9x(56.1h)	0.7x(35.3h)	0.8x	0.9x	1728.81	**2.4x(2.4h)**	1.4x(31.6h)	1.3x(29.2h)	1.1x	1.9x
Geomean(Avg)	-	**1.5x(1.7h)**	1.1x(39.2h)	0.9x(36.5h)	0.9x	1.0x	-	**1.7x(1.7h)**	1.4x(28.5h)	1.5x(27.0h)	1.0x	1.4x

conducted experiments with 2, 4, and 6 thread counts. Based on experimental findings, we determined the optimal number of threads for the best performance.

3.2 Overall Performance for representative deep learning models

The baseline for each model's inference time is established based on the results of ArmNN, and the relative performance was examined in comparison to this baseline. As seen in Table 2, ACLTuner showed a performance that was 1.1x to 1.8x faster than the baseline for all models on Edge R. ACLTuner outperformed other tuners on all models except MobileNetV2. While ACLTuner still improved the performance of MobileNetV2, the library contains only a single kernel for depth-wise convolution, offering no opportunities for tuning by ACLTuner. Experiments on Odroid N2+ also showed similar results to Edge R, with performance improvement of 1.3x to 2.4x. Notable is the fact that ACLTuner provides superior performance for the unique model, ResNetXY.

3.3 Tuning cost

AutoTVM and Ansor utilize black-box optimization techniques to improve performance, without relying on existing kernels. Furthermore, they integrate machine learning-based cost models to provide guidance during the optimization procedure. This feature allows the systems to efficiently determine a suitable execution without the need to manually search through all possible scenarios. ACLTuner, unlike the others, utilizes existing kernels, resulting in a relatively small search space. ACLTuner also incorporate the cost model with the objective of minimizing tuning time. Notably, the performance variance of ACLTuner remains under 2% compared to when conducting an exhaustive search across all conceivable scenarios. As a result, the average tuning time is reduced by 95% and even the overall performance of the result was better.

4 Conclusion and Future Work

Our study proposed ACLTuner, a system designed to optimize an existing library tailored to specific deep learning models and hardware configurations. By employing a cost model with a limited time budget, ACLTuner was able to determine the most effective execution configurations. As a result, we observed a performance enhancement of up to 2.0x compared to the ArmNN baseline. Furthermore, compared to three existing tuners, ACLTuner significantly reduced the tuning time by up to 95%.

ACLTuner currently only supports ArmNN library. As part of future work, we are planning to expand ACLTuner's capabilities to include additional libraries, such as ONNXRuntime and TFLite. This would enable the instrument to access a wider variety of kernels and determine the most efficient execution strategy. In addition, addressing circumstances in which only suboptimal kernels are available is on our agenda.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2023-00277060, Development of open edge AI SoC hardware and software platform, 50%, and No.2022-0-00454, Technology development of smart edge device SW development platform, 50%)

References

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[2] Open source ml compiler. `https://github.com/openxla/`. Accessed: 2023-09-29.

[3] Hsin-I Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. Tinyiree: An ml execution environment for embedded systems from compilation to deployment. *IEEE Micro*, 42(5):9–16, 2022.

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

[5] ONNX Runtime. `https://onnxruntime.ai/`, November 2018.

[6] Intel onednn. `https://github.com/oneapi-src/oneDNN`. Accessed: 2023-09-29.

[7] Nvidia cudnn. `https://developer.nvidia.com/cudnn/`. Accessed: 2023-09-29.

[8] Tensent ncnn. `https://github.com/Tencent/ncnn`. Accessed: 2023-09-29.

[9] Armnn. `https://review.mlplatform.org/admin/repos/ml/armnn`. Accessed: 2023-07-23.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[11] 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.

[12] 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.

[13] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4), jul 2018.

[14] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[16] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems*, 4:204–216, 2022.

[17] Ruochen Hao, Qinglin Wang, Shangfei Yin, Tianyang Zhou, Siqi Shen, Songzhu Mei, and Jie Liu. Towards effective depthwise convolutions on armv8 architecture, 2022.

[18] Tflite. `https://www.tensorflow.org/mobile/tflite/`. Accessed: 2023-09-28.

[19] big.little technology: The future of mobile. `https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf`. Accessed: 2023-09-24.

[20] Asus tinker edge r. https://tinker-board.asus.com/product/tinker-edge-r.html. Accessed: 2023-09-28.

[21] Hardkernel odroid n2+. https://wiki.odroid.com/odroid-n2/odroid-n2. Accessed: 2023-09-28.

[22] Partha P. Maji, Andrew Mundy, Ganesh S. Dasika, Jesse G. Beu, Matthew Mattina, and Robert D. Mullins. Efficient winograd or cook-toom convolution kernel implementation on widely used mobile cpus. *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 1–5, 2019.

[23] Google xnnpack. https://github.com/google/XNNPACK. Accessed: 2023-09-29.

[24] Pytorch mobile. https://pytorch.org/mobile/. Accessed: 2023-09-28.

[25] Halo. https://github.com/alibaba/heterogeneity-aware-lowering-and-optimization. Accessed: 2023-09-28.

[26] Samsung one. https://github.com/Samsung/ONE. Accessed: 2023-09-28.

[27] 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, pages 785–794, New York, NY, USA, 2016. ACM.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[30] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[31] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[32] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[33] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[34] 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.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[36] Open neural network exchange(onnx). https://onnx.ai/, November 2018.

[37] Nvidia cublas. https://developer.nvidia.com/cublas. Accessed: 2023-09-29.

[38] Arm compute library. https://github.com/ARM-software/ComputeLibrary. Accessed: 2023-09-29.

[39] Xiandong Huang, Qinglin Wang, Shuyu Lu, Ruochen Hao, Songzhu Mei, and Jie Liu. Evaluating fft-based algorithms for strided convolutions on armv8 architectures. *Performance Evaluation*, 152:102248, 2021.

[40] Xiandong Huang, Qinglin Wang, Shuyu Lu, Ruochen Hao, Songzhu Mei, and Jie Liu. Numa-aware fft-based convolution on armv8 many-core cpus. In *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 1019–1026, 2021.

[41] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big.little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2020.

[42] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O'Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. Compiling onnx neural network models using mlir, 2020.

[43] torch-mlir. `https://github.com/llvm/torch-mlir`. Accessed: 2023-09-28.

[44] Tensorcomprehensions. `https://github.com/facebookresearch/TensorComprehensions`. Accessed: 2023-09-29.

[45] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. Value learning for throughput optimization of deep learning workloads. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 323–334, 2021.

[46] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 204–216, 2022.

[47] Miguel de Prado, Andrew Mundy, Rabia Saeed, Maurizo Denna, Nuria Pazos, and Luca Benini. Automated design space exploration for optimized deployment of dnn on arm cortex-a cpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(11):2293–2305, 2021.

[48] Marat Dukhan. The indirect convolution algorithm, 2019.

[49] pthread-based thread pool. `https://github.com/Maratyszcza/pthreadpool`. Accessed: 2023-09-29.

[50] Rk3399pro datascheet. `https://rockchip.fr/RK3399Pro%20datasheet%20V1.1.pdf`. Accessed: 2023-09-28.

[51] Rk3399pro datascheet. `https://dn.odroid.com/S922X/ODROID-N2/Datasheet/S922X_Public_Datasheet_V0.2.pdf`. Accessed: 2023-09-28.

A Related Works

A.1 Inference Libraries

Deep learning frameworks such as TensorFlow [34], PyTorch [35], and ONNX [36] heavily depend on inference libraries for the purposes of training and inferring deep learning models. Nvidia GPUs are accompanied by software libraries such as cuDNN [7] and cuBlas [37], which enhance their performance. Similarly, CPUs are supported by software libraries such as oneDNN [6], NCNN [8], and arm's Compute Libraries [38], which provide optimization and functionality.

In deep learning, the operations with the highest computational intensity are ultimately the convolution layers and the fully connected layers. These can be accelerated by various algorithms like Direct-convolution, FFT, and Winograd, and are implemented to match the target hardware. In particular, algorithms implemented for the ARMv8 architecture target are being continuously developed [39, 40, 17]. Efforts are not limited to kernel optimization; there is also a concurrent focus on optimizing scheduling to increase throughput [41].

A.2 Deep Learning Compilers

Deep learning compilers, such as TVM [1], OpenXLA [2], MLIR [4] based compiler projects [42, 43, 3], and Tensor Comprehensions[44], are designed to take models from various deep learning frameworks and transform them into a distinct intermediate representation (IR). Following this transformation, they undergo both hardware-independent and hardware-dependent optimizations to generate code tailored for specific target devices. Some compilers employ auto-tuning techniques, such as AutoTVM[10], Halide Scheduler[12, 13], FlexTensor [14], Rammer [15], and Ansorčitezheng2020ansor to optimize the execution code. They explore the execution configuration space, profiling candidates directly on the devices to identify optimal execution codes. However, this process can be time-consuming, often spanning hours or even days. Consequently, techniques are employed to accelerate this process [45, 46].

A.3 Integration of Compiler and Library.

Recently, Deep learning frameworks, which used to rely solely on deep learning libraries, have started to introduce their own compilers. Pytorch has introduced TorchDynamo to perform the compilation process, and TensorFlow is not only using its existing XLA but also directly employs the compiler by introducing IREE. Traditional deep learning compilers also have limitations in compiling down to the machine code level and actively utilize established deep learning or math libraries. For instance, TVM can generate code using CUBLAS and ACL. Moreover, execution libraries like ONNXRuntime can perform model inference using other inference libraries such as ACL, XNNPACK, or compilers like TVM.

B Determining Rules in Arm Compute Library

B.1 rule-based method to determine the number of kernels and threads for each layer

Arm Compute Library (ACL) determines the algorithm and thread count based on rules during the process of configuring neural network models. For convolution algorithm selection, factors like kernel size, data layout, and tensor size dictate the choice between Winograd, General, Direct convolution, FFT, and Naive algorithms, as illustrated in Table 3 and Algorithm 1. Depending on the chosen algorithm, pre-processing and post-processing might be required. ACL offers NEON kernels for a variety of data types, including f32, bf16, s8, u8, etc. In our work, we utilized only the f32 kernels listed in Table 3 for tuning. For depthwise convolution, regrettably, only a single kernel for fp32 exists, leaving no scope for kernel selection tuning within ACL.

On the other hand, to determine the number of threads, the minimum value of core counts from each CPU cluster is taken into account. For example, Odroid N2+, with four cores in its big cluster and two in its little cluster, creates two threads. In contrast, Edge R has two cores in its big cluster and four in the little cluster. However, like Odroid N2+, it also creates two threads. In such a big.LITTLE architecture, it is infeasible to use all cores concurrently for computations in any given context.

Algorithm 1 The rule for determining convolution algorithm and kernel in Arm Compute Library.

$I \leftarrow$ Input Tensor
$O \leftarrow$ Output Tensor
$W \leftarrow$ Weight (Kernel Filter)

function ADDGEMM( )
    $M \leftarrow$ Table 3
    for m in M **do**
        $Cost \leftarrow$ Estimate(m)
        if $Cost_{Best} > Cost$ **then**
            $Cost_{Best} = Cost$
        end if
    end for
    return $Cost_{Best}$
end function

function GENERAL( )
    AddIm2col()
    AddGemm()
    if Layout = NCHW **then**
        AddCol2Im()
    end if
end function

function WINOGRAD( )
    AddTranspose()
    AddGemm()
    AddTranspose()
end function

function DIRECT( )
    AddGemm()
end function

function CONVOLUTION(I,W,O)
    if (is_General(I,W,O)) or (dilation $\neq$ (1,1)) **then**
        return General()
    else if total_size(I) $> 10^6$ and height_size(W) > 7 **then**
        return Naive()
    else if height_size(W) > 7 and channel_size(I) > channel_size(O) **then**
        return FFT()
    else if channel_size(I) < 16 or $W_{D[W,H]} = 1$ **then**
        return General()
    else if size of W is in $\{3 \times 3, 3 \times 1, 5 \times 1, 7 \times 1\}$ **then**
        return Winograd()
    else if Layout = NHWC and size of W < 4 **then**
        return Direct()
    else
        return General()
    end if
end function

Table 3: The list of GEMM kernel in ACL.

Name	Kernel Size	Memory Arrange	Supported
SGEMM	8x6	Interleaved [47]	-
	8x12	Interleaved	-
MLA	8x4	Hybrid Indirect [48]	-
	4x24	Hybrid Indirect	-
	6x16	Hybrid Indirect	-
SmallK MLA	6x4	Hybrid Indirect	$8 < K \leq 16$ and $N \pmod 4 = 0$
	8x4	Hybrid Indirect	$K \leq 8$ and $N \pmod 4 = 0$

C Other Inference Libraries

With the diversification of hardware and instruction sets, efficiently performing deep learning operations using a single program implementation is not feasible. Consequently, inference libraries provide micro-kernels that vary based on instruction types, workloads, and algorithms, allowing for diverse operational methods tailored to each hardware. These libraries are designed to select the appropriate micro-kernel for a given deep learning model during install time or runtime, based on various hardware information.

Prominently, ArmNN, TFLite, and ONNXRuntime offer dozens of Micro-Kernels for the same operation, varying based on the instruction set. They provide optimal kernel configurations based on internal rules. Additionally, to minimize idle time in computational devices, they offer various scheduling strategies related to workload distribution. Thus, ACLTuner's tuning methodology can also be integrated into TFLite and ONNXRuntime.

C.1 Supproting Arm CPUs in TFLite with XNNPack

In this research, our TFLite experiments employ XNNPack [23] to process deep learning operations and leverage pthreadpool [49] for parallel execution. XNNPACK is tailored as an optimal solution for neural network inferences across ARM, x86, WebAssembly, and RISC-V platforms. It provides a wealth of Micro-Kernels designed for diverse hardware specifications, facilitating acceleration on platforms including Arm NEON, AVX, and SSE. Specifically, the GEMM Kernel crafted in Arm NEON comprises around 131 Micro-Kernels.

While XNNPack is a library comprising highly optimized kernels, it's challenging to ensure the right kernel selection for every target hardware. Furthermore, pthreadpool, which distributes tasks across Micro-Kernels, requires a predetermined number of threads and doesn't consider individual core performance or cache memory, making optimization complex. In Experiment Table 2, Edge R demonstrated its highest average performance with four threads, while Odroid N2+ did so with six threads.

C.2 Supporting Arm CPUs in ONNXRuntime with MLAS

ONNXRuntime interfaces with various hardware acceleration libraries through its adaptable Execution Providers (EP) framework, enabling optimal execution of ONNX models on diverse hardware platforms. This includes libraries such as ArmNN and XNNPACK. However, in our experiments, we utilized the MLAS library, which is natively implemented in ONNXRuntime. MLAS is a computation library that houses processor-optimized GEMM kernels and platform-specific threading codes. A distinguishing feature of ONNXRuntime is its heuristic scheduling algorithm, which takes cache into account and predicts costs to ensure the utilization of all cores.

D Devices: Edge R and Odroid N2+

In this paper, the devices used for the experiments are Edge R [20] and Odroid N2+ [21]. Both of these devices are equipped with Arm big.LITTLE architecture CPUs, and their configurations are shown in Table 4. ACLTuner has been verified to work on various hardware equipped with the

AArch64 Architecture, such as Raspberry Pi 4, Snapdragon 865, etc. Plans are in place to apply it to a range of Cortex and Neoverse series processors as well.

Table 4: Specsheet of used devices

		Edge R [20]	Odroid N2+ [21]
Kernel Version		4.4.194	4.9.337-132
SoC		RK3399Pro [50]	S922X [51]
Arm Version		Armv8a	Armv8a
Memory		4 Gb	4 Gb
Little	Model	A53	A53
	Core	4	2
	Freq.	1.4 Ghz	2.0 Ghz
	L1/D	32 KB	Non Public
	L1/I	32 KB	Non Public
	L2	512 KB	256 KB
Big	Model	A72	A73
	Core	2	4
	Freq.	1.8 Ghz	2.4 Ghz
	L1/D	32 KB	Non Public
	L1/I	32 KB	Non Public
	L2	1024 KB	1024 KB

E ResNetXY

Standard deep learning models often exhibit certain common characteristics. For example, the number of weights might be a multiple of 16, or both the width and height of the kernel might be set to 3. Such design choices are mainly because these models were developed to exploit parallelism effectively on conventional hardware, resulting in higher throughput. However, when models undergo processes like pruning for compression, their structure can change significantly, potentially preventing efficient hardware utilization with the original kernels. In this research paper, we have modified the ResNet50 to form ResNetXY as shown in Figure 5, aiming to demonstrate that ACLTunercan maintain high performance even on such altered models.

F Cost Model

To train the Cost Model, we utilized XGBoost [27]. We distinguish each task based on the following features: the width (W), height (H), and channel (C) of the input, the number (N), width (W), and height (H) of the weights, and padding and stride sizes. For each task, we train four models: GEMM, input pre-processing, output post-processing, and im2col. Notably, for the Gemm model, additional features used in the training include the types of GEMM algorithm, the type of kernel, the number of workloads, and the workload allocated to each core cluster. Table 6 presents the hyperparameters we used.

During the initial phase of training, we train using 50 randomly generated samples. After this, we iteratively profile a total of 5 samples, comprising the top 3 samples recommended by the cost model and 2 randomly generated samples. Through our experiments, we found that by using this approach, the model appears to be sufficiently trained after about 100 samples.

Table 5: Model architecture of ResNetXY and ResNet50

layer name	output size	repeat	ResNetXY	ResNet50
conv1	112x112	*1	14x14, 51	7x7, 64
conv2	56x56	*3	4x4, 77 6x6, 77 1x1, 154	1x1, 64 3x3, 64 1x1, 256
conv3	28x28	*4	4x4, 97 6x6, 97 1x1, 194	1x1, 128 3x3, 128 1x1, 512
conv4	14x14	*6	4x4, 279 6x6, 279 1x1, 558	1x1, 256 3x3, 256 1x1, 1024
conv5	7x7	*3	4x4, 511 6x6, 511 1x1, 1022	1x1, 512 3x3 512 1x1, 2048
FLOPs			16.35 G	4.13 G
Params			86.61 M	25.55 M

Table 6: Hyperparameters

Parameters	Value
Objective	reg:squarederror
Learning_Rate	0.2
Max_Depth	9
Min Child Weight	2
Gamma	0
Reg Alpha	1
Subsample	0.53
Colsample Bytree	0.8
Eval Metric	RMSE