



# Luthier: Bridging Auto-Tuning and Vendor Libraries for Efficient Deep Learning Inference

**YONGIN KWON**, Electronics and Telecommunications Research Institute, Yuseong-gu, Korea (the Republic of)

**JOOHYOUNG CHA**, University of Science and Technology, Daejeon, Korea (the Republic of)

**SEHYEON OH**, University of Science and Technology, Daejeon, Korea (the Republic of)

**MISUN YU**, Electronics and Telecommunications Research Institute, Yuseong-gu, Korea (the Republic of)

**JEMAN PARK**, AI Computing Research Laboratory, Electronics and Telecommunications Research Institute, Yuseong-gu, Korea (the Republic of)

**JEMIN LEE**, Electronics and Telecommunications Research Institute, Daejeon, Korea (the Republic of)

Recent deep learning compilers commonly adopt auto-tuning approaches that search for the optimal kernel configuration in tensor programming from scratch, requiring tens of hours per operation and neglecting crucial optimization factors for parallel computing on asymmetric multicore processors. Meanwhile, hand-optimized inference libraries from hardware vendors provide high performance but lack the flexibility and automation needed for emerging models. To close this gap, we propose Luthier, which significantly narrows the search space by selecting the best kernel from existing inference libraries, and also employs cost model-based profiling to quickly determine the most efficient workload distribution for parallel computing. As a result, Luthier achieves up to 2.0x faster execution on convolution-based vision models and transformer-based language models (BERT, GPT) on both CPUs and GPUs, while reducing average tuning time by 95% compared with ArmNN, AutoTVM, Ansor, ONNXRuntime, and TFLite.

CCS Concepts: • **Computing methodologies** → **Machine learning approaches**; *Parallel algorithms*; • **Computer systems organization** → **Embedded systems**; **System on a chip**;

Additional Key Words and Phrases: deep learning, asymmetric multicore processor, parallel computing

## ACM Reference Format:

Yongin Kwon, JooHyoungh Cha, Sehyeon Oh, Misun Yu, Jeman Park, and Jemin Lee. 2025. Luthier: Bridging Auto-Tuning and Vendor Libraries for Efficient Deep Learning Inference. *ACM Trans. Embedd. Comput. Syst.* 24, 5s, Article 92 (September 2025), 23 pages. <https://doi.org/10.1145/3759916>

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, No.RS-2023-00277060, No.RS-2025-02217404, No.RS-2025-02214497, No.RS-2025-02216517).

Authors' Contact Information: Yongin Kwon, Electronics and Telecommunications Research Institute, Yuseong-gu, Daejeon, Korea (the Republic of); e-mail: yongin.kwon@etri.re.kr; JooHyoungh Cha, University of Science and Technology, Daejeon, Daejeon, Korea (the Republic of); e-mail: jh.cha@etri.re.kr; Sehyeon Oh, University of Science and Technology, Daejeon, Korea (the Republic of); e-mail: osehn@etri.re.kr; Misun Yu, Electronics and Telecommunications Research Institute, Yuseong-gu, Korea (the Republic of); e-mail: msyu@etri.re.kr; Jeman Park, AI Computing Research Laboratory, Electronics and Telecommunications Research Institute, Yuseong-gu, Daejeon, Korea (the Republic of); e-mail: jeman@etri.re.kr; Jemin Lee (corresponding author), Electronics and Telecommunications Research Institute, Daejeon, Daejeon, Korea (the Republic of); e-mail: leejaymin@etri.re.kr.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 1539-9087/2025/09-ART92

<https://doi.org/10.1145/3759916>

## 1 Introduction

During inference, the efficiency of deep learning execution is mainly influenced by deep learning compilers [13, 28, 34, 35, 45] or inference libraries [1, 3, 5, 6, 9, 19]. Given adequate computational resources and time, deep learning compilers generate execution code that is optimized for the specific target hardware. In particular, AutoTVM [45] and Anso [37], during the auto-tuning process, create a diverse range of codes and expertly identify the optimal code to maximize hardware performance, without requiring extensive domain-specific and hardware knowledge.

On the other hand, inference libraries are designed to immediately initiate computational processes on the target device upon receiving an input model. These libraries include highly optimized kernels that handle the computation of the smallest units of deep learning operations. Tailored to specific domains, these kernels are specialized and implemented on a case-by-case basis, leveraging in-depth hardware expertise. During execution, the libraries utilize predefined rules to select appropriate kernels and distribute them to the target hardware to process the entire deep learning operations.

Due to their high flexibility and amenability to automation, compilers are particularly advantageous for the optimal execution of unique and novel deep learning operations. However, exhaustively searching through all code configurations is exceedingly time-consuming. Machine learning-based auto-tuning algorithms [15, 18, 30, 37] are employed to mitigate this time, but it still often takes many hours.

Hardware platforms that share the same **Instruction Set Architecture (ISA)**, such as AArch64, have the capability to execute identical code. This uniformity allows the execution of a single code across various hardware implementations. However, it does not guarantee that an execution optimized for one hardware implementation will be equally optimal for another. Additionally, asymmetric multicore processors have varied performance characteristics, which depend on the specific combination of cores that make up each cluster. To enhance parallel processing performance, it is essential to identify and customize the optimal execution configuration tailored to each specific hardware implementation [47]. This process involves not only finding the best kernel but also determining the most suitable workload distribution for each core. However, to our knowledge, there has been no research in deep learning compilers to date that considers these critical aspects.

To address the challenges discussed, we propose Luthier, an automated inference library tuner that optimizes execution configurations by selecting the most suitable kernels and distributing workloads across asymmetric multicores. The core concept involves leveraging hand-optimized computational kernels provided by hardware vendors. Luthier effectively reduces search space by leveraging these optimized kernels and improves parallel processing performance through the distribution of core-specific workload. As the optimal execution configuration varies with changes in hardware or target operations, Luthier builds a machine learning-based cost model during the tuning process. The model is designed to rapidly identify the most effective execution configuration. To integrate Luthier into ArmNN and XNNPack, we modified only a few hundred lines of code across several files. This low implementation cost enables Luthier to be easily deployed and adapted to vendor-specific computing libraries.

We perform comprehensive experiments comparing Luthier with traditional libraries including ArmNN [3], XNNPack [5], ONNXRuntime [1], and TFLite [10], as well as auto-tuning works Anso and AutoTVM, to demonstrate the reduced optimization search time and improved end-to-end execution time. As a result, Luthier achieves substantial speedups ranging from 1.1x to 1.8x over ArmNN across CNN-based vision models tested on the Edge R, and from 1.3x to 2.4x compared with the ArmNN on the Odroid N2+. Moreover, it demonstrates performance enhancements of up to 1.8x over ArmNN for Transformer-based language models: BERT (encoder-only) and GPT (decoder-only). These results exceed the enhancements achieved by traditional auto-tuning

research frameworks, including AutoTVM and Ansor, which do not consider tuning workload distribution. Interestingly, Luthier also requires 95% less tuning time compared with AutoTVM and Ansor.

To show Luthier's compatibility and scalability, Luthier is also applied to XNNPack, where it exhibited similar performance improvements. Demonstrating hardware platform scalability, Luthier is successfully implemented in big.Little asymmetric multicore environments on Edge R and Odroid N2+, as well as on the Snapdragon 865, configured with three clusters, all of which show enhanced performance. Furthermore, Luthier also applies to the mobile GPU (Mali-G52), demonstrating performance improvements. This indicates that optimizing kernel selection alone is meaningful even in environments with symmetric cores.

The contributions in this article are summarized as follows:

- Luthier is the first library-level auto-tuning framework that simultaneously optimizes kernel selection and cluster-wise workload partitioning through a learning-based cost model while fully reusing vendor hand-tuned kernels. Whereas Ansor already faces a search space of  $\sim 10^{100}$  solely for kernel selection, Luthier requires only 15 kernel candidates and still keeps the combined search space (# of kernels  $\times$  workload distribution) below  $10^6$ .
- We conduct preliminary experiments that highlight the significant performance benefits of optimal workload distribution on asymmetric multicore CPUs, which Luthier efficiently manages by leveraging existing kernel codes in inference libraries crafted with deep hardware and domain knowledge.
- Luthier is implemented with ArmNN and XNNPack through minimal code changes, and rigorously tested on various devices and processors, demonstrating significant performance enhancements of up to a 2.0x speedup for standard vision models and up to a 1.8x speedup for language models.
- We compare Luthier's performance with other solutions, including inference libraries and compilation tuning methods such as AutoTVM and Ansor, showing that Luthier outperforms these alternatives by reducing tuning time by 95% through the incorporation of machine learning to minimize the number of profiling samples.

## 2 Preliminaries

This section presents background and research motivations: Section 2.1 details the structure of inference libraries; Section 2.2 discusses how workload distribution in asymmetric multicore architectures affects performance; Section 2.3 discusses tuning options and search spaces for optimizing inference libraries.

### 2.1 Structure of Inference Libraries

Deep learning models are typically represented as graphs composed of layer nodes interconnected by communication edges. Inference libraries transform these graphs into a series of operators, which are subsequently sorted and executed sequentially.

Figure 1 illustrates the workflow of ArmNN executing convolution operations on an asymmetric multicore architecture. It demonstrates the multitude of choices available at each stage of the process, from selecting appropriate algorithms and kernels to effectively distributing workloads across various core clusters.

For selecting a convolution algorithm, ArmNN under the default setting facilitates the execution of a convolution operation using several algorithms: Winograd [39], Im2col+GEMM [17], FFT [14], and Direct Convolution [40]. Each algorithm is specifically designed to accelerate computations by converting convolution operations into **general matrix multiplications (GEMMs)**, although this transformation can increase memory usage.

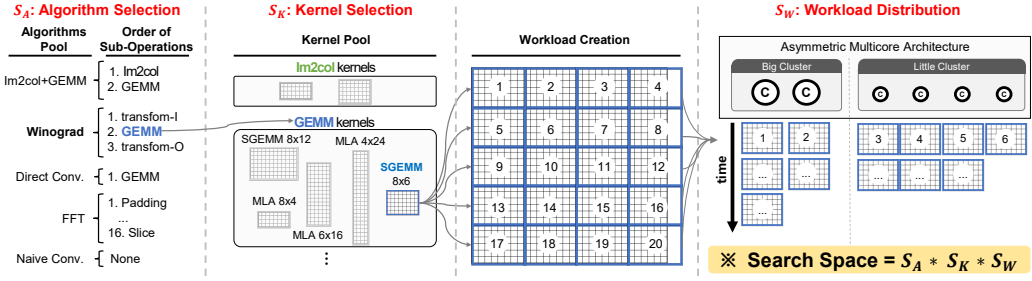


Fig. 1. The workflow of the inference library executing convolution operations on an asymmetric multicore architecture.

---

### ALGORITHM 1: Simplified Heuristic for Convolution Algorithm Selection in ArmNN

---

```

1: function ALGORITHM_SELECTION( $I, W, O$ )
2:   if conditions favor Winograd (e.g., small fixed-size filters) then
3:     return Winograd
4:   else if conditions favor FFT (e.g., large filters, deep channels) then
5:     return FFT
6:   else if layout is NHWC and filter size is small then
7:     return DirectConvolution
8:   else
9:     return Im2col+GEMM or General
10:  end if
11: end function

```

---

The library chooses an appropriate algorithm based on predefined rules. Algorithm 1 details this selection process. The decision-making branches out depending on various factors such as the sizes of the convolution operation’s inputs, weights, outputs, and any dilation settings. Specifically, line 2 of the algorithm highlights that convolutions typically employed in architectures like ResNets and VGGs are preset to be optimal for the Im2col+GEMM algorithm, with a subsequent verification of this fit. If none of the described algorithms are deemed available, the library resorts to a naïve method for performing the convolution.

Each convolution algorithm executes its entire operation by sequentially running the appropriate sub-operations. For example, Direct Convolution involves only a GEMM as its sub-operation, whereas Im2col+GEMM involves two, and FFT requires a total of sixteen sub-operations. Each sub-operation utilizes kernels, which are optimized codes dedicated to the smallest computational units.

In the case of GEMM sub-operation, the library hosts a diverse array of kernels, each developed with distinct algorithms and computational unit sizes to optimize performance. Notably, within ArmNN, these kernels are specialized for specific microarchitectures, such as a53 and a55 CPU cores, and are meticulously implemented in assembly code. Figure 1 shows the five GEMM kernels, which include variations of **Single-precision General Matrix Multiply (SGEMM)**, widely used for general matrix-matrix multiplication in fully connected layers, and **Multiply-Accumulate (MLA)**, often used in CNN inference for combined GEMM, bias, and ReLU operations after im2col transformation. Each kernel differs in shape and size.

Once the appropriate kernel is selected, it becomes clear how many times this kernel needs to be executed to complete the computation for all inputs. These executions are represented as a

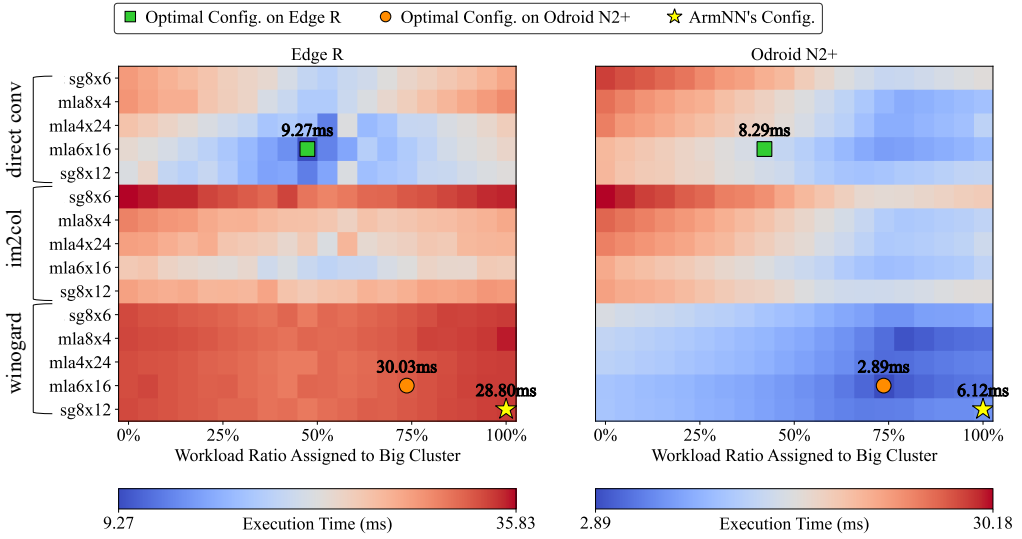


Fig. 2. Heatmap visualization of execution time (ms) for Resnet18's 2<sup>nd</sup> convolution layer on Edge R and Odroid N2+. Each row corresponds to a unique combination of convolution algorithm and kernel, while the X-axis represents the percentage of workload assigned to the big core cluster. Square and circle markers denote optimal configurations for Edge R and Odroid N2+, respectively, and stars represent ArmNN's default setting. The color gradient indicates execution time, with blue representing faster execution.

workload composed of kernel invocations. If there are multiple processing cores available, it is necessary to distribute the workload among these cores to efficiently manage the computations.

In cases with symmetric multicore processors, evenly distributing the workload across cores is more straightforward due to the uniformity of the cores' capabilities. However, in asymmetric multicore processors, it is clear that a different approach to distribution is necessary.

## 2.2 Motivation

Despite possessing highly pre-optimized kernels, inference libraries frequently face the challenge of supporting a wide range of operations and hardware configurations. In addition to this, there is a need to make timely decisions on kernel selection and workload distribution. Consequently, these constraints often lead to suboptimal decisions.

In the context of Arm's big.LITTLE CPU architectures [4], a type of asymmetric multicore processor, the hardware implementation can vary significantly based on the core type and configuration. To validate the importance of kernel selection and workload distribution, we conducted a preliminary experiment assessing the execution time of a convolution operation on two devices equipped with big.LITTLE CPUs: the Edge R [11] and the Odroid N2+ [8].

In these experiments, when using ArmNN to run the second convolution layer of ResNet18 on each device, the rule on both devices selects Winograd for the convolution algorithm and SGEMM\_8x12 for the GEMM kernel. The library configures the threads to match the cluster with the fewest cores and distributes the workload evenly among them.

Additionally, we modified the library to remove the predetermined rules, enabling manual selection of the algorithm, kernel, and workload distributions. Figure 2 presents two heatmaps that illustrate the performance variability depending on the combination of these choices. As shown in Figure 2, the stars indicate the execution times under ArmNN's predefined rules that employ

Table 1. The Example of Tuning Knobs from AutoTVM

Knobs	Description	Possible Value
$tile_{kc}$	Loop tiling parameters on the number of channels, height, and weight of filters	$\{1, \dots, kc\}$
$tile_{kh}$		$\{1, \dots, kh\}$
$tile_{kw}$		$\{1, \dots, kw\}$
$tile_{ic}$	Loop tiling parameters on the number of channels, height, and weight of input feature maps	$\{1, \dots, ic\}$
$tile_{ih}$		$\{1, \dots, ih\}$
$tile_{iw}$		$\{1, \dots, iw\}$
$unroll$	Inner-most loop unroll factor	$\{1, \dots, \}$
$loop\ order$	The order of the six nested loops	6!
$threading$	The number of parallel threads	$\{1, \dots, \}$

the Winograd algorithm and the SGEMM\_8×12 kernel. In these cases, the workload is allocated solely to the big cluster. The circles indicate the best performance on the Odroid N2+, where this configuration outperforms ArmNN (star symbol) on the Odroid N2+ but shows inferior results on the Edge R. Conversely, the squares highlight the best performance scenarios for the Edge R, where these settings improve upon ArmNN (star symbol) configuration on the Edge R but underperform on the Odroid N2+. Notably, this configuration employs the Direct Convolution algorithm, which differs from the predefined rule’s convolution algorithm.

The experimental findings have revealed two key insights: first, the predefined rule-based operation libraries do not yield optimal performance, particularly in terms of algorithm and kernel selection and workload distribution on asymmetric multicore architectures. Second, because an optimal execution configuration for one hardware might not be optimal for another, it is necessary to tailor the tuning for each specific hardware.

This discrepancy arises from differences in microarchitectural characteristics such as the number and type of cores, frequency, instruction throughput, and memory hierarchy between Edge R and Odroid N2+. Table 4 highlights these architectural differences.

### 2.3 Tunable Knobs and Search Spaces

In preceding sections (Sections 2.1 and 2.2), it was shown that execution configurations based on predefined rules fall short of achieving the best performance, indicating a significant opportunity for optimization.

Therefore, there is a necessity to adjust execution depending on the specific deep learning operation and the target device. These adjustable parameters, referred to as knobs, enable the exploration of all possible combinations within the search space.

Deep learning compilers like TVM [45] optimize deep learning operations, which often consist of multiple loops, by applying techniques such as tiling and unrolling. This process supports backend compilers like LLVM in optimally allocating registers and scheduling instructions, which in turn directly generates execution code that aligns with the kernels of the inference libraries.

AutoTVM [45] employs a template-based tuning approach, setting up predefined tuning knobs as illustrated in Table 1. These knobs essentially dictate the sequence, size, and parallelism of a kernel’s operations through methods such as tiling and unrolling. By adjusting these values, AutoTVM searches for the optimal code within a defined space, with the search space in this instance estimated to be around  $10^{13}$ . In contrast, Ansor [37] does not predetermine templates but instead continuously performs tuning tasks such as tiling and reordering to discover the optimal code. The search space for Ansor is significantly larger, estimated to extend to approximately  $10^{100}$  [32].

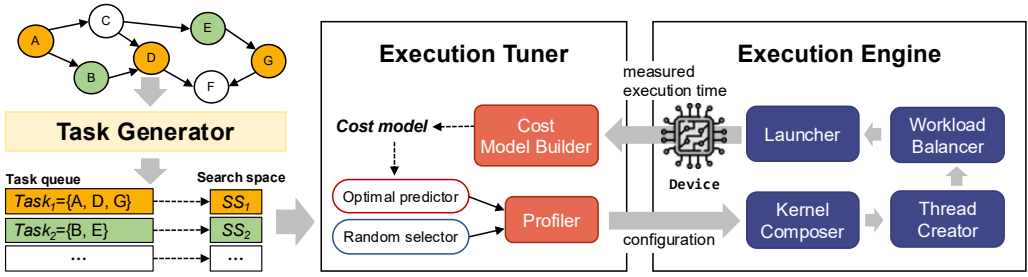


Fig. 3. Overall tuning process of Luthier.

On the other hand, inference libraries lack adjustable knobs for kernel modification. They feature a limited set of kernel implementations, which are already finely tuned in the assembly language for the target hardware by skilled developers. The scope of tuning within these libraries is limited to selecting the most appropriate algorithm and kernel. As shown in Figure 1,  $S_A$  and  $S_K$  denote algorithm selection and kernel selection, respectively.

The experimental results showcased in Figure 2 have established that workload distribution profoundly influences the performance of deep learning inference on asymmetric multicore processors. Consequently, it is crucial to treat workload distribution as a tuning knob. Specifically for GEMM operations, the execution might generate potentially thousands of kernel invocations. The performance outcomes vary not just based on the number of the invocations each CPU cluster handles, but also on the specific invocations assigned and the sequence of their execution. Such complexity can potentially increase the overall search space to beyond  $10^{15}$ . Nevertheless, by implementing various heuristics, such as presuming that workloads are evenly distributed among cores within the same cluster, this expansive search space can be significantly narrowed down to approximately  $10^6$  ( $S_W$  in Figure 1).

Consequently, the total search space is expressed as  $S_A \times S_K \times S_W$ , reflecting the combined dimensions of algorithm selection, kernel selection, and workload distribution within the search parameters. This results in a considerably smaller search space compared with that of AutoTVM and Ansor.

Thus, in the subsequent parts of the article, we detail the development of the concept of Luthier designed to navigate the vast search space efficiently. This involves selecting the most suitable kernels and allocating workloads optimally to find the ideal execution configuration.

### 3 Luthier: Optimizing Compute Libraries for Diverse Hardware

Luthier is designed to automatically identify the optimal kernel and workload distribution with minimal profiling on real devices. To achieve this objective, Luthier consists of a *Task Generator*, an *Execution Tuner*, and an *Execution Engine*, as shown in Figure 3 and Algorithm 2.

**Task Generator** generates a task queue with tasks that require tuning from the operations of the input model. During this process, deep learning operations that have identical input, output, and weight dimensions are grouped together to form a single tuning task (Lines 7–12), thus reducing the time required for tuning. For example, in the case of ResNet101, although there are initially 104 operations requiring tuning, the process of removing duplicates effectively narrows it down to just 23 tasks remaining in the task queue.

In Luthier, each task is associated with a specific search space, which plays a crucial role in the tuning process. For convolution operations, there are three main tuning knobs: algorithm, kernel, and workload distribution, as detailed in Table 2. Conversely, for fully connected operations, which

---

**ALGORITHM 2:** The Overall Algorithm of the Task Generator and the Execution Tuner for Tuning a Given Input Model  $M$ 


---

```

1:  $TaskQueue \leftarrow \{\}$  ▷ Initialize Task Queue buffer
2:  $TunedTask \leftarrow \{\}$  ▷ Initialize TunedTask dictionary
3:  $BestConf \leftarrow \{\}$  ▷ Initialize BestConf dictionary
4:
5: function TASKGENERATOR( $M$ )
6:   for  $op$  in  $M$  do
7:     if !FINDTASK( $op, TaskQueue$ ) then
8:        $task \leftarrow$  CREATETASK( $op$ )
9:        $TaskQueue \leftarrow TaskQueue \cup \{task\}$ 
10:    else
11:       $task \leftarrow$  FINDTASK( $op$ )
12:    end if
13:     $TunedTask[op] \leftarrow task$ 
14:  end for
15: end function
16:
17: function EXECUTIONTUNER( $TaskQueue, \alpha, \beta$ )
18:   for  $task$  in  $TaskQueue$  do
19:      $costModel \leftarrow \phi$  ▷ Model to predict top N optimal configurations
20:      $best \leftarrow \phi$  ▷ perf: Performance such as execution time
21:      $confs \leftarrow$  RANDOM( $\alpha$ ) ▷ confs: Configurations to test on real hardware
22:      $perfs \leftarrow$  EXECUTIONENGINE( $confs$ ) ▷ Run on device to measure actual perf
23:      $costModel \leftarrow$  TRAINMODEL( $confs, perfs, costModel$ ) ▷ Train model using confs and perfs
24:     while TopNChanged( $costModel$ ) do
25:        $confs \leftarrow$  RANDOM( $0.1 \times \alpha \times (1 - \beta)$ ) ▷ Generate random confs
26:        $confs \leftarrow confs \cup$  PREDICT( $costModel, 0.1 \times \alpha \times \beta$ ) ▷ Predict optimal confs
27:        $perfs \leftarrow$  EXECUTIONENGINE( $confs$ ) ▷ Run on device to measure actual perf
28:        $costModel \leftarrow$  TRAINMODEL( $confs, perfs, costModel$ ) ▷ Train model to extract optimal confs
29:        $best \leftarrow$  min( $best, perfs$ ) ▷ Update best conf
30:     end while
31:      $BestConf[task] \leftarrow best$ 
32:   end for
33: end function
34:
35: function EXECUTIONENGINE( $confs$ )
36:    $results \leftarrow \phi$ 
37:   for  $conf$  in  $confs$  do
38:      $kernels \leftarrow$  KERNELCOMPOSER( $conf$ )
39:     ( $bigTHD, littleTHD$ )  $\leftarrow$  CREATETHREAD( $conf$ )
40:     WORKLOADBALANCER( $bigTHD, littleTHD, kernels, conf$ )
41:      $results \leftarrow results \cup$  LAUNCHER( $bigTHD, littleTHD$ )
42:   end for
43:   return  $results$ 
44: end function

```

---

do not require an algorithm adjustment, only the kernel selection and workload distribution serve as tunable knobs.

The choices for the algorithm and kernel knobs are relatively straightforward, providing clear options that can be methodically evaluated and implemented. However, the workload

Table 2. The Tunable Knobs of Luthier

Knobs	Description	Possible Values
<i>algorithm</i>	The convolution algorithm selected to generate the execution kernel. Only one algorithm is used per configuration.	{Winograd, Direct, FFT, ...}
<i>kernel</i>	The specific low-level GEMM microkernel used to execute matrix multiplication. This includes both the instruction type (e.g., ‘mla’, ‘dot’) and the tile shape (e.g., $8 \times 4$ ).	{mla8x4, mla4x4, dot8x8, ...}
<i>cluster workload distribution</i>	Distribution of total workload across clusters (e.g., big, middle, little cores). The values assigned to each cluster must be non-negative integers, and their sum must equal the total number of tiles to compute.	$[w_{big}, w_{mid}, w_{lit}]$ , e.g., [8, 4, 4]

distribution knob introduces a much higher level of complexity, considerably expanding the search space.

**Execution Tuner** plays a key role in determining the best execution configuration for each task in the task queue by directly running them on the device. Given the extensive search space, exhaustively exploring all potential execution configurations presents a significant challenge. To address this, a *cost model builder* is used to develop a *cost model* for each task. This model is designed to predict the execution time given a configuration, which includes a specific deep learning operation and its tuning knob parameters. The input features for the cost model include the selected algorithm (e.g., Winograd, FFT), kernel (e.g., mla8x4, sg8x12), and workload distribution across CPU clusters (e.g., Refs. [8, 2, 2]), as shown in Table 2. Categorical features such as algorithm and kernel are one-hot encoded, and the workload distribution is represented as a normalized vector. A *profiler* then uses the *cost model* to effectively navigate the vast search space, focusing on the most promising execution configurations. By accurately forecasting execution times, the cost model enables the identification of the most efficient execution configuration with minimal need for extensive profiling. The details of the process are as follows.

- ① The *Execution Tuner* begins by retrieving the next task from the *task queue* (Line 18). Each task corresponds to a group of operations in the model that share identical dimensions, allowing tuning to be conducted more efficiently. For the selected task, the tuner explores execution configurations by profiling both random and cost-model-guided candidates.
- ② Then, the *Random Selector* (shown in Figure 3) generates  $\alpha$  random configurations, each of which is executed on the device to collect performance data. This data is used to train the first version of the *cost model* (Lines 21–23).
- ③ In each subsequent iteration, the *profiler* evaluates a fixed number ( $0.1 \times \alpha$ ) of configurations, composed of both randomly selected and model-predicted candidates. While the random candidates are again provided by the *Random Selector*, the model-guided ones are selected by the *Optimal Predictor* (shown in Figure 3). The ratio between random and predicted configurations is governed by parameter  $\beta$  (Lines 25–27). The measured execution times are then used to incrementally update the cost model (Line 28), improving its prediction accuracy and preventing overfitting to the initial samples.
- ④ This iterative tuning continues until the top- $N$  configurations predicted by the model stabilize, indicating convergence (Line 24). Once convergence is detected, the optimal configuration is recorded (Line 31), and the tuner moves on to the next task from the queue, repeating the above process.

To develop the *cost model*, various machine learning algorithms can be used such as regression, decision tree, random forest, and gradient boosting. Each of these approaches offers advantages and disadvantages in terms of accuracy and computational cost. Luthier primarily employs XGBoost [44] due to its superior performance in handling large datasets and its computational efficiency. Since the goal is to predict execution time, the cost model is trained as a regression task minimizing the RMSE between the predicted and actual execution times. This loss metric is also used as the convergence criterion during tuning (Line 24).

**Execution Engine** computes deep learning operations using the execution configurations passed by the *profiler* during the tuning process (Lines 22 and 27), or it can operate independently after the tuning process is complete. Additionally, the *Execution Engine* sequentially executes a series of components: a *kernel composer*, a *thread creator*, and a *workload balancer*.

The *kernel composer* within the *Execution Engine* combines one or more kernels to customize them for the specified target operation (Line 38). For instance, the Direct Convolution algorithm is configured using only a single GEMM sub-operation, thus constituting one GEMM kernel as determined by the execution configuration. In contrast, the Im2col+GEMM configuration involves both the im2col sub-operation and the GEMM sub-operation, leading to the composition of two distinct kernels.

The *thread creator* is responsible for generating dedicated threads for each CPU cluster. The execution configuration includes the workload distribution ratio between big cluster and little cluster for each kernel, and it generates threads that match the number of cores in the clusters to be used (Line 39).

The *workload balancer* is tasked with distributing workload to each thread according to the execution configuration (Line 40). To streamline the process, it ensures that threads within the same cluster receive an equal amount of sub-tasks. The execution configuration contains information about the proportion of workload to be handled by each cluster, and the *workload balancer* distributes the workload to each thread in accordance with these proportions.

The *launcher* reads the thread information, then executing these threads accordingly. After the execution, it collects the output and the execution time. This data is then passed back to the *Execution Tuner*. The information provided by the launcher is essential for training the *cost model* (Line 41).

Once the tuning is finalized, the optimal execution configuration is saved (Line 31). This optimized configuration is then directly implemented by the *Execution Engine* for future operations, thereby bypassing the need for further tuning by the *Execution Tuner*.

### 3.1 Implementation

The proposed algorithm is designed to be easily integrated into widely used deep learning inference libraries. To apply Luthier, it is sufficient to perform minimal modifications to specific execution paths within each library, such as internal operation scheduling, kernel calls, thread creation, and distribution. Due to this simplicity, Luthier can effectively reuse existing kernels, which are already highly optimized for specific hardware or platforms, enabling it to easily generate optimal execution code.

Table 3 shows the number of **Source Lines of Code (SLOC)** changed and the number of files modified when integrating Luthier into different libraries. Most modifications involve connecting Luthier's tuning logic (kernel selection, thread allocation, load balancing) to existing kernel invocation points within each library. As clearly shown in Table 3, integrating Luthier typically requires only minimal code changes compared with the total size of the original libraries. For example, integrating Luthier with *ArmNN (CPU)* required only 400 changed lines of code and modifications in 9 files out of the total 943K lines of code and 5K files in the original library. These

Table 3. Source Lines of Code (SLOC) and Number of Files Changed When Integrating Luthier Into Existing Computing Libraries

	ArmNN (GPU)	ArmNN (CPU)	XNNPack (CPU)
<b>Changed SLOC / Total SLOC</b>	382 / 943K	400 / 943K	527 / 7,318K
<b>Modified Files / Total Files</b>	11 / 5K	9 / 5K	12 / 24K

The relatively small amount of code modification required highlights the ease of adopting Luthier in various inference libraries.

Table 4. CPU Cluster Configurations and Cache Hierarchies of Edge R, Odroid N2+, and Snapdragon 865

Cluster	Specification	Edge R	Odroid N2+	Snapdragon 865
Prime	Microarchitecture	–	–	Kryo 585 Gold
	Core Count	–	–	1
	Frequency	–	–	2.84 GHz
	L1 Cache (I/D)	–	–	64 KB / 64 KB
	L2 Cache	–	–	512 KB
	L3 Cache	–	–	4 MB
Big	Microarchitecture	Cortex-A72	Cortex-A73	Kryo 585 Gold
	Core Count	2	4	3
	Frequency	1.8 GHz	2.4 GHz	2.42 GHz
	L1 Cache (I/D)	48 KB / 32 KB	Not Disclosed	64 KB / 64 KB
	L2 Cache	1 MB	1 MB	256 KB
Little	Microarchitecture	Cortex-A53	Cortex-A53	Kryo 585 Silver
	Core Count	4	2	4
	Frequency	1.4 GHz	2.0 GHz	1.8 GHz
	L1 Cache (I/D)	32 KB / 32 KB	Not Disclosed	32 KB / 32 KB
	L2 Cache	512 KB	256 KB	128 KB

results demonstrate that Luthier can be incorporated without needing substantial restructuring of existing frameworks or rewriting kernels. Consequently, Luthier does not require extensive reimplementations at the compiler or runtime level and can easily extend support to new hardware or emerging deep learning models.

The main reason why Luthier can be integrated with minimal modifications is its design, which directly reuses existing hardware-specialized kernels while only invoking Luthier’s tuning logic at runtime. This allows dynamic selection of optimal kernels and workload distribution without having to write new kernels or manually perform complex optimizations, maintaining the high performance and compatibility of the original libraries.

## 4 Experiments

### 4.1 Experimental Setup and Implementation

The initial implementation of Luthier was developed using ArmNN, and we have since expanded its functionality to include support for XNNPack. We have modified the inference libraries to remove predefined rules, enabling dynamic selection and execution of kernels based on the given execution configuration. Additionally, we have implemented a static workload distribution by assigning processor affinity to each thread, ensuring that computations are confined to specific cores.

We performed experiments on representative deep learning vision models, including ResNet18, ResNet50, ResNet101, AlexNet, VGG16, GoogLeNet, and MobileNetV2. To further validate the robustness of Luthier with non-standard architectures, we developed ResNetXY, a variant of ResNet

Table 5. Search Space Size (Number of Candidate Configurations) for Each Model and Device

Model	Edge R	Odroid N2+	SD865
ResNet18	$2.304 \times 10^5$	$2.304 \times 10^5$	$7.974 \times 10^7$
ResNet50	$1.098 \times 10^6$	$1.098 \times 10^6$	$2.167 \times 10^9$
ResNet101	$2.020 \times 10^6$	$2.020 \times 10^6$	$3.611 \times 10^9$
AlexNet	$7.129 \times 10^4$	$7.129 \times 10^4$	$2.721 \times 10^7$
VGG16	$1.139 \times 10^6$	$1.139 \times 10^6$	$6.215 \times 10^9$
GoogLeNet	$3.842 \times 10^5$	$3.842 \times 10^5$	$2.075 \times 10^8$
MobileNetV2	$3.341 \times 10^5$	$3.341 \times 10^5$	$3.873 \times 10^8$
ResNetXY	$6.951 \times 10^5$	$6.951 \times 10^5$	$8.311 \times 10^8$

in which the number and sizes of kernels in each layer are randomly selected from those used in ResNet50. Furthermore, our experimental scope extended to modern transformer-based language models, including BERT-base (encoder-only) and GPT2-small (decoder-only).

The experiments are carried out mainly on two edge devices featuring asymmetric multicore CPUs, a Tinker Edge R and an Odroid N2+. Detailed specifications are presented in Table 4. We further carried out experiments to show the effectiveness of Luthier on a 3-cluster CPU and a mobile GPU. This includes Snapdragon 865, which is equipped with three types of Kryo CPU cores, and Arm Mali-G52 MP6 GPU on the Odroid N2+.

Luthier was tested against ArmNN (v22.05), AutoTVM (v0.10), AnsoR (v0.10), ONNXRuntime (v1.15.1), and TFLite (v2.14) with XNNPack delegate. AutoTVM and AnsoR have tuning processes similar to Luthier, while the others execute inference directly, without any tuning process. To train the cost models, we utilized XGBoost (v2.0.3), the same tool employed by TVM for training regression models.

During the initial training phase, Luthier utilizes 50 randomly generated samples, referred to as  $\alpha$  in Algorithm 2. Subsequently, the training process involves iterative profiling of 5 samples per iteration, which includes the top 3 samples suggested by the cost model and 2 samples generated randomly, indicating a  $\beta$  value of 0.4. Our experiments indicate that this method allows the model to be adequately trained after approximately 100 samples. We set the value of  $N$  to 50 when checking the stability of the top- $N$  predicted configurations to determine convergence of the tuning process.

To quantitatively characterize the tuning complexity of Luthier, Table 5 presents the size of the search space for each model on different target devices. The search space is defined as the total number of possible execution configurations explored by Luthier, which includes combinations of algorithm selection, kernel selection, and workload distribution across CPU clusters.

Edge R and Odroid N2+ show identical search space sizes because both have dual-cluster CPUs with a total of six cores, leading to the same number of workload distribution options. In contrast, the Snapdragon 865's tri-cluster structure (1 big, 3 medium, 4 little) yields a much larger search space due to more partitioning combinations, which Luthier efficiently handles via its learning-based tuning process.

## 4.2 Performance Enhancement on Deep Learning Operations

Figure 4 presents a performance comparison of GEMM operations for multiplying two  $N \times N$  matrices on the Edge R, with  $N$  varying from 64 to 1,792. The results demonstrate that when  $N$  is less than 128, the performance of Luthier closely matches that of the Little Cluster Only configuration. However, as  $N$  exceeds 128, the performance of Luthier begins to significantly improve, eventually stabilizing at approximately 50 GFLOPS. In contrast, the performance of the

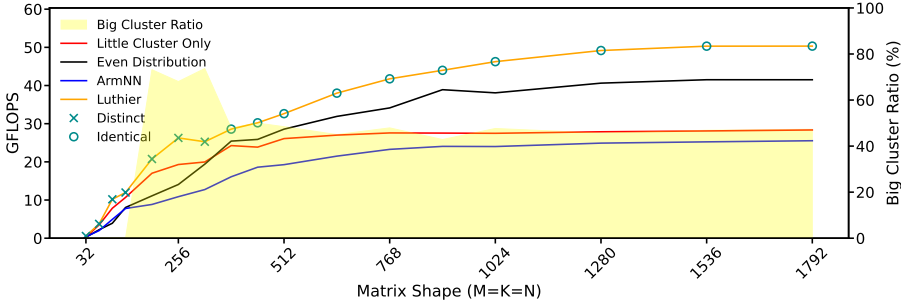


Fig. 4. GFLOPS performance for the multiplication of two square matrices ( $N \times N$  shape) on Edge R, using both the ArmNN library and Luthier separately. It also illustrates the performance variations when workload distribution differs across each cluster. On the Luthier line, a circle notation indicates the selection of a kernel that matches one chosen by predefined rules in ArmNN, while a cross notation represents selections that do not follow these rules.

Little Cluster Only remains nearly constant at around 27 GFLOPS, while ArmNN, which utilizes only two CPUs from the big cluster, remains at approximately 23 GFLOPS.

Luthier selects a different type of kernel than ArmNN for  $N < 384$ , but converges to the same kernel choice as ArmNN for  $N \geq 384$ . This indicates that the performance improvement in this region is primarily attributed to optimized workload distribution rather than kernel selection.

As  $N$  increases, the computation intensity also increases, making the operation more likely to become compute-bound rather than memory-bound. When  $N < 300$ , the configuration using only the four cores in the little cluster (excluding Luthier) achieves the highest performance. This aligns with the observation that the little cores cease to provide further gains once the computation becomes the bottleneck around  $N = 300$ . Beyond this point, configurations that utilize both big and little cores begin to exhibit increasing performance. At  $N = 1,536$ , the performance of all methods starts to converge. Notably, the performance of Luthier roughly matches the sum of ArmNN and the little-cluster-only configurations, indicating that Luthier effectively leverages both clusters to their full potential.

Figure 5 shows how Luthier enhances the performance of convolution operations compared with ArmNN and XNNPack, and how it stands relative to AutoTVM and Ansor. Not surprisingly, the results for Luthier show a geometric mean speedup of 2.8x over ArmNN and 2.6x over XNNPack. Moreover, in most cases, Luthier surpasses the performance of AutoTVM and Ansor, demonstrating its effectiveness and efficiency in optimizing deep learning convolution operations.

Despite extensive tuning efforts, AutoTVM and Ansor cannot match the performance of Luthier primarily because they lack the capability to tune workload distribution effectively; they focus solely on tuning kernel code. In TVM, parallelization is one of the tuning knobs, leading to the creation of multiple sub-tasks. Ideally, these sub-tasks should be parallelized to the extent possible, but the actual distribution decision is handled by TVM's runtime. The runtime, similar to inference libraries, generates a predetermined number of threads based on fixed rules, typically corresponding to the number of big cores on asymmetric multicore processors. Additionally, the assignment of workloads to these threads is determined by modulating the sub-task index, which inherently limits flexibility in workload distribution—preventing dynamic or uneven distribution among the threads. Therefore, AutoTVM and Ansor fall short in performance compared with Luthier, which finely adjusts workload distribution between both big and little clusters of an asymmetric multicore processor, optimizing resource use and improving overall efficiency.

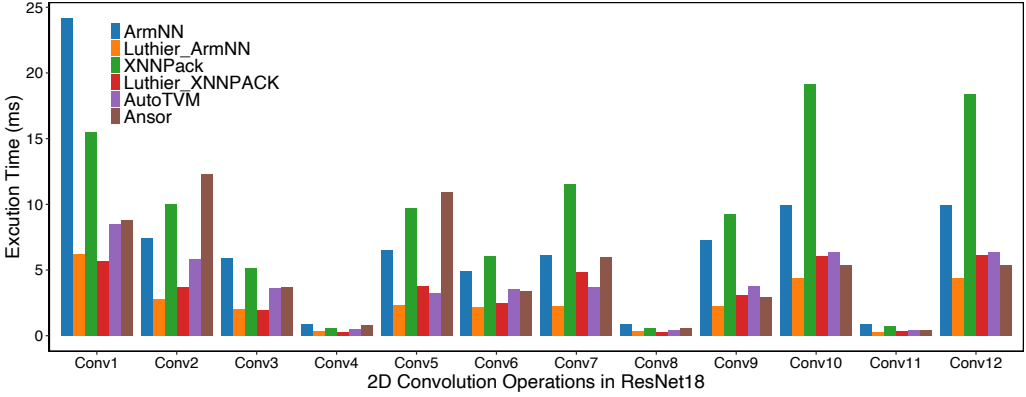


Fig. 5. Layer-wise performance comparison for convolution layers in ResNet18 on Odroid N2+.

Table 6. End-to-End Performance Comparison on the CPUs of Edge R. The Baseline is the Inference Time (ms) with ArmNN

Model	ArmNN (ms)	Luthier	AutoTVM	Ansor	ONNXRuntime	TFLite
ResNet18	195.66	<b>1.4x(1.6h)</b>	1.3x(27.1h)	0.8x(29.2h)	0.9x	1.0x
ResNet50	480.23	<b>1.6x(2.0h)</b>	1.2x(19.5h)	1.1xx(27.1h)	1.0x	1.1x
ResNet101	873.11	<b>1.6x(2.7h)</b>	1.1x(51.5h)	1.0x(46.2h)	0.8x	1.1x
AlexNet	124.36	<b>1.6x(0.4h)</b>	0.8x(33.4h)	0.6x(32.7h)	1.0x	1.2x
VGG16	915.24	<b>1.5x(0.9h)</b>	1.0x(33.3h)	0.5x(30.4h)	0.6x	0.6x
GoogLeNet	210.26	<b>1.4x(2.4h)</b>	1.3x(49.8h)	1.1x(28.5h)	1.2x	1.1x
MobileNetV2	118.02	1.1x( <b>1.6h</b> )	1.7x(43.2h)	<b>1.8x</b> (62.7h)	1.5x	1.2x
ResNetXY	1782.50	<b>1.8x(2.4h)</b>	0.9x(56.1h)	0.7x(35.3h)	0.8x	0.9x
Geomean(Avg)	-	<b>1.5x(1.7h)</b>	1.1x(39.2h)	0.9x(36.5h)	0.9x	1.0x

We compared the results of Luthier with AutoTVM, Ansor, ONNXRuntime, and TFLite. Compared with the baseline, the speedup is represented as a factor (x). For Luthier, AutoTVM, and Ansor, which require tuning, the tuning time is denoted in hours (h) and is provided in parentheses.

Table 7. End-to-End Performance Comparison on the CPUs of Odroid N2+

Model	ArmNN (ms)	Luthier	AutoTVM	Ansor	ONNXRuntime	TFLite
ResNet18	138.64	1.7x( <b>1.6h</b> )	<b>1.9x</b> (10.1h)	1.4x(26.0h)	0.9x	1.3x
ResNet50	389.63	<b>2.0x(1.9h)</b>	1.8x( 6.2h)	1.5x(27.1h)	1.1x	1.7x
ResNet101	651.63	<b>2.0x(2.6h)</b>	1.7x(23.2h)	1.4x(34.7h)	0.9x	1.6x
AlexNet	78.47	1.3x( <b>0.4h</b> )	0.6x(53.7h)	<b>1.4x</b> (25.8h)	0.8x	1.1x
VGG16	701.27	<b>1.6x(0.8h)</b>	1.2x(31.0h)	0.8x(25.8h)	0.7x	0.9x
GoogLeNet	147.21	1.6x( <b>2.3h</b> )	<b>1.6x</b> (22.6h)	1.5x(23.2h)	1.2x	1.6x
MobileNetV2	83.02	1.5x( <b>1.5h</b> )	1.8x(49.8h)	<b>2.8x</b> (23.8h)	1.3x	1.4x
ResNetXY	1728.81	<b>2.4x(2.4h)</b>	1.4x(31.6h)	1.3x(29.2h)	1.1x	1.9x
Geomean(Avg)	-	<b>1.7x(1.7h)</b>	1.4x(28.5h)	1.5x(27.0h)	1.0x	1.4x

### 4.3 Overall Performance for Representative Deep Learning Vision Models

Tables 6 and 7 present the end-to-end inference performance, measured as the total execution time of executing all layers in the model graph, including convolution and fully-connected layers. The baseline for inference time of each model is set using the results from original ArmNN, which then

Table 8. Comparative Execution Times for BERT-Base and GPT2-Small Models on Odroid N2+ Across Various Sequence Lengths (SL)

Model	Seq.	execution time (ms)	
		ArmNN	Luthier
BERT-base	64	1,260	1,022 (1.2x)
BERT-base	128	1,782	1,203 (1.5x)
BERT-base	256	2,905	1,801 (1.6x)
BERT-base	384	4,100	2,366 (1.7x)
BERT-base	512	5,383	2,977 (1.8x)
GPT2-small	64	1,400	1,187 (1.2x)
GPT2-small	128	1,922	1,369 (1.4x)
GPT2-small	256	3,020	1,933 (1.6x)
GPT2-small	384	4,272	2,576 (1.7x)
GPT2-small	512	5,731	3,322 (1.7x)

The speedup achieved by Luthier is indicated with a multiplier (x) next to each execution time.

serves as a reference for evaluating the relative speedups. The table shows that Luthier achieves a speed improvement ranging from 1.1x to 1.8x faster than the baseline across all models tested on the Edge R. Notably, Luthier surpasses other tuners in performance for all models except MobileNetV2.

While Luthier enhances the performance of MobileNetV2 compared with ArmNN, it is constrained because it does not offer a better kernel for depth-wise convolution than those found by AutoTVM and Anso. Consequently, despite the optimizations possible through workload distribution, Luthier faces limitations in surpassing the performance enhancements achieved by these other tuners, which have access to more efficient convolution kernels.

Experiments conducted on the Odroid N2+ also mirrored the results seen on the Edge R, demonstrating performance improvements ranging from 1.3x to 2.4x. Particularly noteworthy is the exceptional performance of Luthier with the unique model, ResNetXY. This outcome illustrates Luthier’s robust capability to efficiently handle and optimize new and unconventional model architectures.

#### 4.4 Performance Enhancement for Transformer-Based Language Models

Table 8 provides the performance of Luthier on transformer-based language models, specifically BERT-base<sup>1</sup> (encoder-only) and GPT2-small<sup>2</sup> (decoder-only), using the Odroid N2+ device. The tests vary the sequence length from 64 to 512 to examine how Luthier compares to ArmNN in terms of execution time.

The results indicate that at a sequence length of 64, Luthier achieves a 1.2x speedup over ArmNN. This speedup increases to as much as 1.8x at a sequence length of 512. In these models, the GEMM operations consume more than half of the end-to-end execution time. Notably, when the sequence length reaches 512, over 80% of the entire computational effort is dedicated to executing GEMM operations. Furthermore, similar to the observations in Figure 4, as the matrix size increases—which correlates with increasing sequence length—Luthier’s performance continues to improve significantly.

<sup>1</sup>BERT-base at <https://huggingface.co/google-bert/bert-base-uncased>

<sup>2</sup>GPT2-small at <https://huggingface.co/openai-community/gpt2>

Table 9. Luthier’s Speedup (x) Over XNNPack (ms) for the CPUs of Odroid N2+ and Sanpdragon (SD) 865

Model	Odroid N2+		SD 865	
	XNNPack	Luthier	XNNPack	Luthier
ResNet18	114	1.4x	50	1.4x
ResNet50	254	1.2x	114	1.2x
ResNet101	389	1.0x	223	1.3x
AlexNet	67	1.2x	32	1.4x
VGG16	693	1.0x	434	1.4x
GoogLeNet	78	1.1x	43	1.1x
MobileNetV2	42	1.9x	14	1.6x
ResNetXY	851	1.1x	544	1.4x

#### 4.5 Evaluation of Luthier’s Compatibility on Different Inference Libraries and Hardware Platforms

To demonstrate that Luthier is not limited to specific hardware platforms, we demonstrate its effectiveness through a Qualcomm Snapdragon 865 (SD865) development board. The SD 865 is notable for its DynamIQ [12] architecture, featuring three distinct CPU clusters: big, middle, and little. These clusters are detailed in Table 4.

TensorFlow Lite, PyTorch Mobile, Alibaba’s HALO, and Samsung’s ONE are just a few of the notable frameworks and platforms that use the well-known inference library XNNPack. The internal architecture of XNNPack has similarities to that of ArmNN, featuring a wide variety of kernels. XNNPack and ArmNN both distribute tasks evenly across threads, but XNNPack stands out with its capability to dynamically offload workloads. This feature allows finished threads to assist slower ones by taking over their tasks. However, as Table 9 shows, superior performance is achieved through Luthier’s effective initial workload distribution, which reduces the necessity for later workload offloading.

Our comparative study primarily identified minor performance variations between Luthier when used with ArmNN and XNNPack. However, certain models exhibited significant differences. For instance, Luthier performed considerably better with ArmNN for VGG16, achieving 438.3 ms (a 1.6x speedup) as shown in Table 7, compared with 693 ms (a 1.0x speedup) with XNNPack, as listed in Table 9. In contrast, Luthier was more effective with XNNPack for MobileNet V2, achieving 22.1 ms compared with 55.3 ms with ArmNN. We attribute these performance disparities to the types and richness of kernels provided by each inference library.

#### 4.6 Performance Enhancement on Mobile GPU

ArmNN is an inference library compatible with both Arm CPUs and Arm Mali GPUs. It utilizes nine optimized OpenCL kernels for the GPUs, which are selected based on predefined runtime rules. Unlike the big.LITTLE CPU architecture, the Mali GPU employs a homogeneous multicore architecture. Consequently, we performed a comprehensive search on the Odroid N2+’s Mali-G52 GPU without distributing the workload, a process that takes just a few seconds per model.

As detailed in Table 10, our findings show negligible performance differences for some popular models. However, substantial performance improvements of 2.8x and 2.3x were achieved for AlexNet and ResNetXY, respectively. This shows that significant performance gains can be realized in heterogeneous core environments through optimization of algorithms and kernel selection alone. Considering that these optimizations are completed within seconds, it indicates that it is worthwhile to attempt to optimize.

Table 10. Luthier’s Speedup (x) Over ArmNN for the Mali GPU of Odroid N2+

Model	ArmNN (ms)	Luthier
ResNet18	92	1.0x
AlexNet	219	2.8x
GoogLeNet	109	1.2x
MobileNetV2	56	1.1x
ResNetXY	3479	2.3x

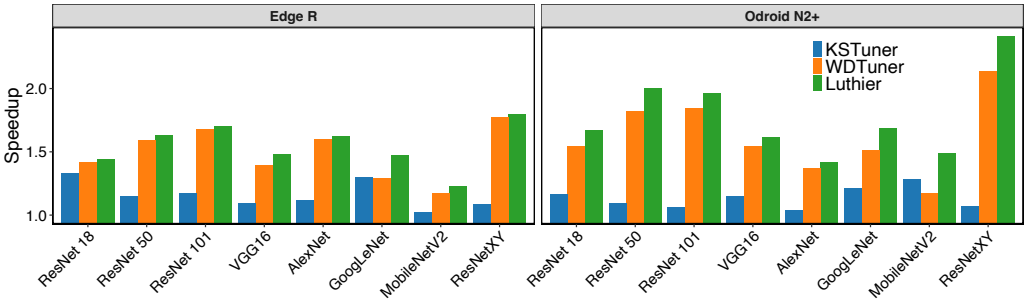


Fig. 6. Speedup achieved by Kernel Selection (KS) Tuner, Workload Distribution (WD) Tuner, and Luthier relative to the ArmNN baseline.

#### 4.7 Comparing Impact of Tuning Knobs on Performance

Luthier optimizes both kernel selection and workload distribution simultaneously, which collectively define the search space, as depicted in Table 2. Focusing solely on one of these aspects, specifically kernel selection, can greatly simplify the search space. Figure 6 shows the influence of each tuning method on Luthier’s performance enhancement. These findings highlight that in most scenarios, optimizing workload distribution has a more significant impact on performance improvement than just selecting the optimal kernel. However, it is noteworthy that kernel selection alone can still lead to substantial performance gains. Interestingly, in the MobileNetV2 experiment on the Odroid N2+, the effect of kernel selection was more pronounced than workload distribution, and the combined tuning of both aspects resulted in a performance improvement greater than the sum of individual enhancements.

#### 4.8 Tuning Performance and Cost

AutoTVM and Ansor utilize machine learning-based cost models to efficiently generate tensor programs. Similarly, Luthier integrates the cost model but is differentiated by operating with fewer tuning knobs. We conducted an experiment in the same environment as described in Section 2.2. Figure 7 displays how Luthier performs with different tuning algorithms: *XGBoost*, *Bayesian*, *Greedy*, and *Random* search. This experiment was repeated five times, and the average performance was recorded. The results indicate that XGBoost consistently achieves optimal execution time by identifying the best execution configuration. In contrast, other search algorithms tend to show slower improvements in execution time or exhibit performance saturation.

As shown in Tables 6 and 7, Luthier significantly reduces the average tuning time by 95%, while also enhancing the overall performance. Furthermore, Figure 8 provides a detailed view of the tuning progress made by Luthier, AutoTVM, and Ansor across four convolution layers from ResNet.

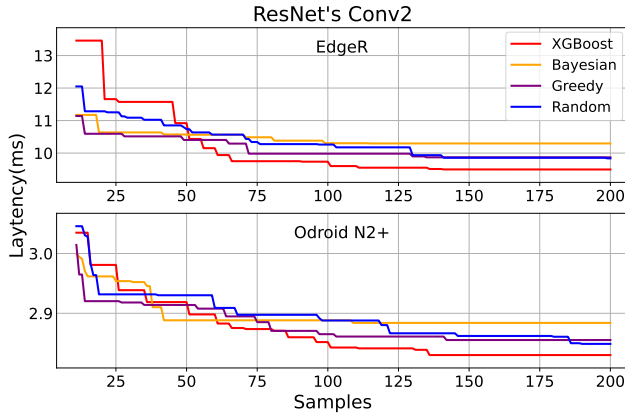


Fig. 7. Comparison of tuning performance according to four search algorithms: Random, Greedy, Bayesian, and XGBoost.

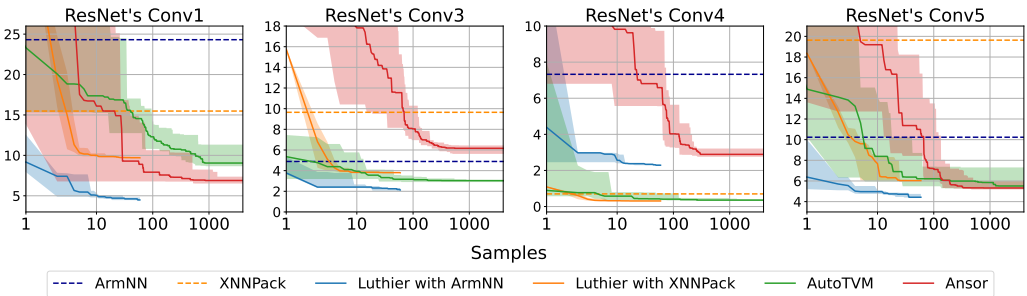


Fig. 8. Comparison of the tuning progress of Luthier with ArmNN and XNNPack against the tuning progress of AutoTVM and Ansor, and the baselines of ArmNN and XNNPack.

Notably, the cost model of Luthier achieved convergence earlier and exhibited greater stability than those used by AutoTVM and Ansor.

## 5 Related Works

### 5.1 Inference Libraries and Runtime Power Management Methods

Deep learning frameworks such as TensorFlow [38], PyTorch [33], and ONNX [2] 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 [19] and cuBLAS [7], which enhance their performance. Similarly, CPUs are supported by software libraries such as oneDNN [6], NCNN [9], and ArmNN, 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 [21–23]. Efforts are not limited to kernel optimization; there is also a concurrent focus on optimizing scheduling to increase throughput [41].

Previous studies on runtime resource management for heterogeneous mobile SoCs [16, 25–27] used online learning methods such as imitation learning and reinforcement learning to optimize DVFS, cluster power gating, and scheduling policies. In contrast, Luthier works at the offline compile-time level. It generates static code by selecting the best kernel algorithm for each layer and deciding how to distribute workloads across clusters. Since these two approaches operate at different time scales, combining optimized code from Luthier with runtime power management methods can lead to additional benefits.

## 5.2 Deep Learning Compilers and Auto-Tuning with Cost Models

Many DNN frameworks and compilers, including AutoTVM [18], Halide Scheduler [29, 46], FlexTensor [42], Ansor [37], Rammer [36], and MetaSchedule [31], employ auto-tuning strategies to search for the tensor implementation with optimal performance. To support such efforts, TenSet [49], a large-scale benchmark dataset for learned tensor compilers, has been introduced, enabling robust cost model training and fair comparison across autotuning approaches. They explore the execution configuration space, profiling candidates directly on devices to identify optimal execution codes. These methods allow for the creation of high-performance code tailored to new deep learning models by starting the search from scratch for each tensor program. To minimize search times, accurate pre-trained cost models are developed, and these models are used to streamline the generation of optimal codes. Recently, research has also focused on adapting cost models to changes in models or hardware to further minimize search times [20, 30, 43, 48]. However, these studies can still be time-consuming, often spanning hours or even days, and the outcomes of tensor programs may not attain performance comparable to that of hardware-native kernels provided by vendors.

## 6 Discussion

This section discusses the extensibility of Luthier and its potential fallback strategies, focusing on how Luthier can be improved to handle more realistic deployment environments.

### 6.1 Considering Execution Time and Power in Extending Luthier

In real mobile and edge environments, there is an inherent tradeoff between execution time and power consumption. Recent studies [16, 25–27] have proposed using online learning methods to adjust system-level control knobs such as DVFS, power gating, and runtime scheduling for multi-objective optimization. However, these methods are often limited in scenarios with strict real-time or battery constraints due to runtime measurement overhead and convergence time.

Luthier currently focuses on optimizing for execution time through offline exploration. Extending it to also consider power introduces several challenges. First, power measurement is costly in terms of time and noise. Measuring energy at the layer or kernel level often requires external equipment or **Performance Monitoring Unit (PMU)** event collection, and the short execution time of these units can result in high measurement errors.

Second, the design space becomes much larger. Luthier currently limits its search space to about  $10^6$  combinations by selecting from 15 kernel candidates and deciding workload distribution. Adding energy as another objective requires identifying a Pareto front, making it difficult to use a simple weighted sum function such as Equation (1) to effectively explore the space.

$$\text{Cost}(\mathbf{c}) = \alpha \text{Execution\_Time}(\mathbf{c}) + (1 - \alpha) \text{Energy}(\mathbf{c}). \quad (1)$$

Since the results are highly sensitive to the value of  $\alpha$ , a more flexible strategy that reflects user preferences is needed.

Third, building accurate predictive models becomes more complex. While execution time tends to be stable, energy consumption varies with factors such as DVFS state, temperature, and memory

access patterns. Therefore, using separate models for each metric and combining their outputs may be more practical than training a single model for both.

Considering these challenges, we identify two practical directions for extending Luthier. First, instead of relying on physical power measurements, we can use hardware counter-based estimation models to get usable energy values without extra equipment. Second, the search strategy can be applied in stages rather than optimizing both metrics at once—for example, narrowing down candidates based on execution time first and then selecting energy-efficient options within that set.

Overall, this study have focused on optimizing execution time, and a multi-objective extension that includes power was left unexplored due to hardware constraints and the absence of reliable measurement infrastructure. Nevertheless, because Luthier keeps its search space sufficiently compact, we anticipate that lightweight predictive models combined with a simple multi-stage search strategy can efficiently extend the framework to jointly optimize both execution time and energy in future work.

## 6.2 Integrating Multiple Vendor Libraries

Luthier currently assumes a single computation library for both optimal kernel selection and workload distribution optimization. Therefore, constructing a unified search space that combines both ArmNN and XNNPack could potentially lead to improved execution time. This kind of extension is not technically infeasible.

In fact, ArmNN and XNNPack are already supported together in runtime environments such as TensorFlow Lite delegates and ONNX Runtime EPs. For instance, ONNX Runtime has supported both the ArmNN EP and the XNNPack EP, and TFLite allows using XNNPack as the default CPU backend while loading ArmNN as an additional delegate. However, achieving integration at the operator level—where different backends are mixed within a single execution graph—requires addressing several implementation challenges.

First, the kernel structure differs significantly. ArmNN implements `im2col` and GEMM as separate kernels, while XNNPack fuses them into a single monolithic kernel. As a result, even for the same convolution operation, the two engines use different data layouts and invocation mechanisms, requiring additional data transformations and logic between kernel boundaries.

Second, memory management becomes complex. Transferring intermediate activations between the two backends demands new buffer allocation, synchronization, and tracking mechanisms. These changes affect core execution paths, increasing maintenance overhead and engineering risk.

Third, the search space expands rapidly. Allowing backend selection at the layer level leads to a combinatorial increase in candidate execution plans, which can quickly exceed realistic auto-tuning budgets.

These challenges are practical rather than fundamental. With future development of a unified memory management layer and lightweight support for data layout transformations, fine-grained cross-backend exploration could become feasible. This presents a promising direction for future work building on Luthier.

## 6.3 Fallback for Unsupported Kernels in Vendor Libraries

Luthier mainly relies on computation libraries. Therefore, if a given operator is not supported by the underlying library, it may suffer from poor performance. To address this issue, Luthier can be extended to use custom kernels generated by tools like Ansor or AutoTVM as a fallback when high-performance kernels are not available in the library.

Although this fallback mechanism is not currently implemented, it can be integrated into Luthier at the layer level through the following steps. First, we can register multiple backends using frameworks like ONNX-Runtime Execution Provider or TFLite Delegate, and explore different

combinations of ArmNN, XNNPack, and Anso for each layer. Second, we can expand the design space to include all layer and backend candidates, and include the backend type as an input feature for the cost model.

A related system that demonstrates this kind of flexibility is Collage [24], one of the first systems to mix multiple deep learning backends at the operator level. Collage improves overall model performance by choosing the best backend for each operation, depending on their characteristics and performance differences. While Collage focuses on backend-level integration, Luthier focuses on kernel selection and core allocation within each backend, making them complementary approaches.

This fallback strategy allows Luthier to maintain its workload partitioning and prediction accuracy even under the assumption of a single backend, while practically extending its coverage for operators not supported by the main library.

## 7 Conclusion

Our study proposed Luthier, a system designed to optimize an existing library tailored to specific deep learning models on asymmetric multicore processors. By employing a cost model with a limited time budget, Luthier was able to determine the most effective execution configurations. As a result, we observed a CPU performance enhancement of up to 2.0x for standard models compared with the original inference libraries. Furthermore, since even recent and customized models such as LLMs continue to rely on the same basic kernel operations used in our experiments, we believe that Luthier can also be effectively applied to optimize their inference performance.

Moreover, while AutoTVM and Anso are known for identifying highly optimized kernels, Luthier has demonstrated superior performance on asymmetric multicore processors and has significantly reduced the tuning time by up to 95%. Integrating Luthier with these deep learning compilers could potentially lead to even greater performance enhancements and support for a variety of hardware, including deep learning accelerators.

## References

- [1] 2018. ONNX Runtime. Retrieved from <https://onnxruntime.ai/>
- [2] 2018. Open Neural Network Exchange(ONNX). Retrieved from <https://onnx.ai/>
- [3] 2023. ArmNN. Retrieved from <https://review.mlplatform.org/admin/repos/ml/armnn>
- [4] 2023. big.LITTLE Technology. Retrieved from <https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf>
- [5] 2023. Google XNNPack. Retrieved from <https://github.com/google/XNNPACK>
- [6] 2023. Intel oneDNN. Retrieved from <https://github.com/oneapi-src/oneDNN>
- [7] 2023. NVIDIA cuBLAS. Retrieved from <https://developer.nvidia.com/cublas>
- [8] 2023. Odroid N2+. Retrieved from <https://wiki.odroid.com/odroid-n2/odroid-n2>
- [9] 2023. Tensent NCNN. Retrieved from <https://github.com/Tencent/mncnn>
- [10] 2023. TFLite. Retrieved from <https://www.tensorflow.org/mobile/tflite/>
- [11] 2023. Tinker Edge R. Retrieved from <https://tinker-board.asus.com/product/tinker-edge-r.html>
- [12] 2024. DynamIQ. Retrieved from <https://www.arm.com/technologies/dynamiq>
- [13] 2024. XLA. Retrieved from <https://github.com/openxla/xla>
- [14] Tahmid Abtahi, Colin Shea, Amey Kulkarni, and Tinoosh Mohsenin. 2018. Accelerating convolutional neural network with FFT on embedded hardware. *IEEE Transactions on Very Large Scale Integration Systems* 26, 9 (2018), 1737–1749.
- [15] Byung Hoon Ahn, Sean Kinzer, and Hadi Esmaeilzadeh. 2022. Glimpse: Mathematical embedding of hardware specification for neural compilation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1165–1170.
- [16] Toygun Basaklar, A. Alper Goksoy, Anish Krishnakumar, Suat Gumussoy, and Umit Y. Ogras. 2023. DTRL: Decision tree-based multi-objective reinforcement learning for runtime task scheduling in domain-specific system-on-chips. *ACM Transactions on Embedded Computing Systems* 22, 5s (2023), 1–22.
- [17] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Suvisoft.
- [18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. arXiv:1805.08166. Retrieved from <https://arxiv.org/abs/1805.08166>

- [19] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. Cudnn: Efficient primitives for deep learning. arXiv:1410.0759. Retrieved from <https://arxiv.org/abs/1410.0759>
- [20] Perry Gibson and José Cano. 2022. Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 28–39.
- [21] Ruochen Hao, Qinglin Wang, Shangfei Yin, Tianyang Zhou, Siqi Shen, Songzhu Mei, and Jie Liu. 2022. Towards Effective Depthwise Convolutions on ARMv8 Architecture. arXiv:2206.12124. Retrieved from <https://arxiv.org/abs/2206.12124>
- [22] Xiandong Huang, Qinglin Wang, Shuyu Lu, Ruochen Hao, Songzhu Mei, and Jie Liu. 2021. Evaluating FFT-based algorithms for strided convolutions on ARMv8 architectures. *Performance Evaluation* 152 (2021), 102248.
- [23] Xiandong Huang, Qinglin Wang, Shuyu Lu, Ruochen Hao, Songzhu Mei, and Jie Liu. 2021. 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*. 1019–1026.
- [24] Byungsoo Jeon, Sunghyun Park, Peiyuan Liao, Sheng Xu, Tianqi Chen, and Zhihao Jia. 2022. Collage: Seamless integration of deep learning backends with automatic placement. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 517–529.
- [25] Sumit K Mandal, Ganapati Bhat, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y Ogras. 2020. An energy-aware online learning framework for resource management in heterogeneous platforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 25, 3 (2020), 1–26.
- [26] Sumit K Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y Ogras. 2019. Dynamic resource management of heterogeneous mobile platforms via imitation learning. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 12 (2019), 2842–2854.
- [27] Gaurav Narang, Aryan Deshwal, Raid Ayoub, Michael Kishinevsky, Janardhan Rao Doppa, and Partha Pratim Pande. 2023. Dynamic power management in large manycore systems: A learning-to-search framework. *ACM Transactions on Design Automation of Electronic Systems* 28, 5 (2023), 1–21.
- [28] Jeman Park, Misun Yu, Jinse Kwon, Junmo Park, Jemin Lee, and Yongin Kwon. 2024. NEST-C: A deep learning compiler framework for heterogeneous computing systems with artificial intelligence accelerators. *ETRI Journal* 46, 5 (2024), 851–864. DOI: <https://doi.org/10.4218/etrij.2024-0139>
- [29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530.
- [30] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. 2022. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 89–103.
- [31] Junru Shao, Xiyu Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor program optimization with probabilistic programs. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*. 35783–35796.
- [32] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value learning for throughput optimization of deep learning workloads. In *Proceedings of Machine Learning and Systems*, Vol. 3. 323–334.
- [33] Adam Paszke et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems* 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035.
- [34] Chris Lattner et al. 2021. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.
- [35] Hsin-I Liu et al. 2022. TinyIREE: An ML execution environment for embedded systems from compilation to deployment. *IEEE Micro* 42, 5 (2022), 9–16.
- [36] Lingxiao Ma et al. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897.
- [37] Lianmin Zheng et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.
- [38] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and others. 2016. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.
- [39] Partha P. Maji et al. 2019. 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* (2019), 1–5.
- [40] Sergio Barrachina et al. 2023. Reformulating the direct convolution for high-performance deep learning inference on ARM processors. *Journal of Systems Architecture* 135 (2023), 102806.

- [41] Siqi Wang et al. 2020. High-throughput CNN inference on embedded ARM Big.LITTLE multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2254–2267.
- [42] Size Zheng et al. 2020. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.
- [43] Shanjun Zhang et al. 2022. FamilySeer: Towards optimized tensor codes by exploiting computation subgraph similarity. arXiv:2201.00194. Retrieved from <https://arxiv.org/abs/2201.00194>
- [44] Tianqi Chen et al. 2016. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA). ACM, 785–794.
- [45] Tianqi Chen et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.
- [46] Tzu-Mao Li et al. 2018. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.* 37, 4, Article 139 (jul 2018), 13 pages.
- [47] Simon Wunderlich, Juan A. Cabrera, Frank H. P. Fitzek, and Martin Reisslein. 2017. Network coding in heterogeneous multicore IoT nodes with DAG scheduling of parallel matrix block operations. *IEEE Internet of Things Journal* 4, 4 (2017), 917–933. DOI : <https://doi.org/10.1109/JIOT.2017.2703813>
- [48] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 833–845.
- [49] Lianmin Zheng, Ruo Chen Liu, Junru Shao, Tianqi Chen, Joseph E. Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Received 24 July 2025; revised 24 July 2025; accepted 29 July 2025