Lorien: Efficient Deep Learning Workloads Delivery

Cody Hao Yu
Amazon Web Services, Inc
hyuz@amazon.com

Xingjian Shi
Amazon Web Services, Inc
xjshi@amazon.com

Haichen Shen
Amazon Web Services, Inc
shaichen@amazon.com

Zhi Chen
Amazon Web Services, Inc
chzhi@amazon.com

Mu Li
Amazon Web Services, Inc
mli@amazon.com

Yida Wang
Amazon Web Services, Inc
wangyida@amazon.com

ABSTRACT

Modern deep learning systems embrace the compilation idea to self generate code of a deep learning model to catch up the rapidly changed deep learning operators and newly emerged hardware platforms. The performance of the self-generated code is guaranteed via auto-tuning frameworks which normally take a long time to find proper execution *schedules* for the given operators, which hurts both user experiences and time-to-the-market in terms of model developments and deployments.

To *efficiently deliver a high-performance schedule upon requests*, in this paper, we present Lorien, an open source infrastructure, to tune the operators and orchestrate the tuned schedules in a systematic way. Lorien is designed to be extensible to state-of-the-art auto-tuning frameworks, and scalable to coordinate a number of compute resources for its tuning tasks with fault tolerance. We leveraged Lorien to extract thousands of operator-level tuning tasks from 29 widely-used models in Gluon CV model zoo [22], and tune them on x86 CPU, ARM CPU, and NVIDIA GPU to construct a database for queries. In addition, to deliver reasonably high performance schedules for unseen workloads in seconds or minutes, Lorien integrates an AutoML solution to train a performance cost model with collected large-scale datasets. Our evaluation shows that the AutoML-based solution is accurate enough to enable *zero-shot tuning*, which does not fine-tune the cost model during tuning nor perform on-device measurements, and is able to find decent schedules with at least 10× less time than existing auto-tuning frameworks.

1 INTRODUCTION

As the wide adoption of deep learning becomes a trend in many domains, the performance of deep learning models becomes crucial. To achieve high performance for inference tasks, major deep learning frameworks (e.g., TensorFlow [1], PyTorch [32], and MXNet [10]) leverage optimized kernel libraries (e.g., cuDNN [13], OneDNN [23]) provided by hardware vendors to accelerate commonly used deep learning operators. However, it is hard for kernel libraries to keep up with the rapid emergence of new operators and hardware cloud/edge platforms.

On the other hand, deep learning compilers, such as Halide [35], XLA [45], Tensor Comprehensions [42], and TVM [11], directly generate operator kernels for different hardware platforms. This enables operator developers to quickly sketch the semantics of a new operator using a high-level declarative language and evaluate the end-to-end model accuracy. The performance of the new operators can be caught up later by either a hardware expert or automatic schedule tuning frameworks [2, 12, 31, 39, 42, 46, 47]. As a result, this approach is more scalable and can shorten the time-to-market compared to the kernel libraries.

Although auto-tuning frameworks are capable of delivering high-performance operators that match or even beat vendor kernel libraries, auto-tuning a deep learning model could take days or even weeks, especially for the model with many workloads like ResNet-152 or Inception V3. A workload is defined as an operator (e.g., `Conv2D` or `Dense`) or a subgraph (e.g., `Conv2D - BiasAdd - ReLU`) with certain values of attributes (e.g., data/weight shapes, strides, padding, and data type), and tuning one workload requires to construct schedules, tune parameters, and perform on-device measurements. This process usually takes one or few hours. For example, according to our experience, using AutoTVM [12] to tune all workloads in ResNet-50 needs 10 hours on x86 CPUs, 7 days on NVIDIA GPUs, and 10 days on Raspberry Pi 4. Even worse, as we will evaluate in subsection 5.1, *one-for-all* schedules do not exist, meaning that an efficient schedule of a workload on a hardware platform is usually not efficient

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3486973

on another platform. As a result, all workloads have to be tuned on every target hardware platform.

To maintain the best user experience during deep model developments and deployments, one key question is: *How to promptly deliver schedules with reasonably good performance upon user requests?* To achieve this goal, we need to tune a large amount of deep learning workloads from deep learning model zoos (i.e., MLPerf [36], Gluon CV model zoo [22], TensorFlow hub [20], and PyTorch torchvision [17]) on many hardware devices. We then commit the best schedule of each workload to a database and query them in milliseconds when needed. However, this is challenging for the following reasons.

Tuning Process Scalability and Stability. Long tuning time affects not only the time-to-market but the stability. To the best of our knowledge, none of existing auto-tuning frameworks is designed for tuning on multiple machines, and none of them consider fault tolerance[1]. The tuning process, hence, has to be *manually started over* if it was accidentally interrupted. This is crucial especially on edge devices, which are less reliable than cloud instances and may fail frequently due to overheat or other factors.

Tuning Result Management. Although almost all auto-tuning frameworks, such as Halide auto-scheduler [2] and AutoTVM [12], provide mechanisms to serialize tuning results for future applications, all of them use file-based mechanism and have different formats. As a result, engineers have additional work to orchestrate the data for efficient usage.

Time to Deliver an Efficient Schedule. Even a database is constructed to serve most user requests, it is still possible that certain workloads are missing. For example, neural architecture search (NAS) may generate unseen workloads. This necessitates the tuning of workloads on-the-fly using the auto-tuning framework. However, modern auto-tuning frameworks usually leverage iterative search algorithms with on-device measurements, which usually take hours, to find an efficient schedule for an unseen workload. The unfavorably expensive querying/tuning overhead makes production deployment impractical.

To address these challenges, we design and implement Lorien, a unified and extensible open source infrastructure to orchestrate the tuning of deep learning workloads at scale. Lorien serves as an abstraction layer between auto-tuning deep learning frameworks and compute resources, such as cloud (e.g., Amazon EC2 [3]) and edge (e.g., self-hosted device farms) platforms, to significantly improve the auto-tuning throughput and efficiency. Lorien abstracts mandatory components in auto-tuning frameworks as high-level APIs, therefore, state-of-the-art auto-tuning frameworks are allowed to be easily plugged in as a dialect. Lorien provides a distributed system to tune a large amount of tuning tasks from vary auto-tuning frameworks on Amazon EC2 instances [3] or edge devices, with the consideration of scalability, flexibility, and reliability. The tuned schedules are committed to a database. Lorien designs a general data model that can accommodate tuning results from various auto-tuning frameworks. We have leveraged Lorien to extract thousands of operator-level tuning tasks from 29 widely-used models in Gluon CV model zoo [22], and tune them on x86 CPU, ARM CPU, and NVIDIA GPU to construct a database for queries. To the best of our knowledge, this is the largest database of deep learning workload schedules.

In addition, in case the user-requested workload has no tuned schedules in the database, Lorien performs *zero-shot tuning*[2] to deliver a decent schedule in a reasonable time. However, most performance cost models adopted by existing auto-tuning frameworks are not designed for the zero-shot tuning and usually operate on heavily-engineered low-level hardware features. Different from these systems, the performance cost model in Lorien is trained on high-level scheduling features via automated machine learning (AutoML). With extensive experiments, we demonstrated that our solution, which is built on top of AutoGluon [16] and is trained on a large-scale scheduling database, is able to obtain highly accurate cost model that can support zero-shot tuning. Our evaluation shows that Lorien is guaranteed to deliver the schedule achieving 80+% performance against the auto-tuning frameworks in seconds to a few minutes upon requests.

In summary, this paper makes the following contributions:

- We design and implement an extensible and reliable distributed infrastructure, Lorien, to tune and orchestrate more than a thousand deep learning tuning workloads with billions of schedule candidates each on cloud platforms and edge devices, and construct the largest database of deep learning workload schedules.
- We employ AutoML to train a performance cost model based on the collected schedules at scale for highly accurate zero-shot tuning.
- We conduct a number of evaluations on top of Lorien to show the effectiveness of Lorien to deliver good schedule on time and reveal a few interesting observations, such as *one-for-all* schedules do not exist, and

[1] Although AutoTVM serializes tuning results to a local file timely so users can manually skip tuned tasks when relaunching the tuning process from failure, it requires a certain level of understanding to AutoTVM.

[2] Zero-shot tuning means the performance cost model is not fine-tuned with the measured schedules during the tuning process. It implies no actual compilation and on-device measurement in tuning.

zero-shot tuning models are only possible upon a large-scale dataset.

We open-source Lorien at https://github.com/awslabs/lorien/ to call for collaboration to further expand the database and share the knowledge with the community.

2 BACKGROUND

Many new deep learning models with novel architectures as well as operators have been proposed by scientists in recent years to achieve high accuracy. In order to deploy up-to-date deep learning models to plenty hardware platforms, including CPUs, GPUs, FPGAs, and ASICs, flexible code generation with performance auto-tuning mechanism becomes a trend for deep learning compilers [2, 12, 31, 39, 42, 46, 47].

A common programming model adopted by these compilers was originally proposed by Halide [35] for image processing workloads. The programming model decouples the description of an operator to a "computation" and a "schedule". The former defines the functionality of an operator in a mathematical representation; while the latter indicates how this computation should be executed on a hardware platform. In particular, a schedule is composed of a series of transformations to rewrite a program while guaranteeing the mathematical equivalent.

One significant advantage of this programming model is that a scientist can quickly implement the computation of a new operator to evaluate the end-to-end model accuracy without worrying about the execution performance, which can be caught up later by a hardware expert or an auto-tuning framework with proper schedules for the target hardware platform. We categorize auto-tuning frameworks to two classes based on their approaches.

Template-based auto-tuning frameworks, such as AutoTVM [12] and FlexTensor [47], leverage operator-based schedule templates written by domain experts. Specifically, from a deep learning model, template-based frameworks extract operators and map them to the corresponding schedule templates as *operator-level* tuning tasks. A tuning task is composed of a workload and tuning configurations (e.g., the hardware platform, tuning time, and so on). Then, they use the proposed tuning algorithms to sequentially search for the best schedule parameters of each tuning task.

The advantages of this approach are three-fold. First, even a workload appears multiple times in a model, there will be just one tuning task. For example, we observed 40% redundant operators from 29 Gluon CV models [22], meaning that the total tuning time can be reduced by 40%. Second, all tuning tasks can be tuned in parallel, which enables more opportunities to improve the auto-tuning in scale. Third, the log of a tuning task can be applied to the same operator in other deep learning models. For instance, after studying the models from Gluon CV model zoo, PyTorch torchvision [17], and TensorFlow hub [20], we observed that the operator overlapping ratio of the same model (e.g., ResNet-50) implemented in different model zoos can range from 52% to 83%.

Auto-scheduling frameworks, such as Tensor Comprehensions [42] and Halide auto-scheduler [2], generates schedules from scratch. Given a deep learning model, they directly analyze the model graph to generate schedule candidates for tuning. Accordingly, the generated schedules are at the *model level* and tightly-coupled with a model, i.e. they can only be applied to the model it was generated from.

Besides, another state-of-the-art auto-scheduling framework, Ansor [46], partitions the model graph into several subgraphs based on certain heuristic rules, and auto-schedules each subgraph separately. This approach has two major advantages: First, all subgraphs can be scheduled in parallel. Second, the generated schedules are at the *subgraph level* and can be shared with the identical subgraph in another deep learning model.

Lorien supports tuning tasks from auto-tuning frameworks in both categories. We will detail Lorien in the next section.

3 LORIEN INFRASTRUCTURE

Although state-of-the-art deep learning auto-tuning frameworks are designed and implemented for a certain deep learning compiler, they share the same framework architecture design as shown in Figure 1. Specifically, given a deep learning model file, the framework first identifies tuning tasks, which can be in any granularity as described in the previous section. Then, it tunes the identified tasks using a tuning algorithm. The algorithm is usually guided by an evaluation metric, which can be an analytical or machine learning performance cost model, on-device measurement, or the combination of both. All explored schedules will be maintained in a storage, which can be a text file or a database. Finally, the deep learning compiler queries the best schedule from the storage and compiles the deep learning model to be a deployable binary file. Lorien is designed as an extensible infrastructure that abstracts the mandatory components in auto-tuning frameworks as high-level APIs, which allow any auto-tuning frameworks to be plugged in as dialects. As a result, Lorien is able to allocate their tuning tasks to the available computing resources, monitor the tuning process, and automatically recover the failure tasks.

Figure 2 depicts the overall Lorien infrastructure. First of all, Lorien employs a *command line interface(CLI)* to accept a command line string or a YAML file so that it interacts with the external requests. With the CLI, a simple string-based message passing protocol between the tuning master and

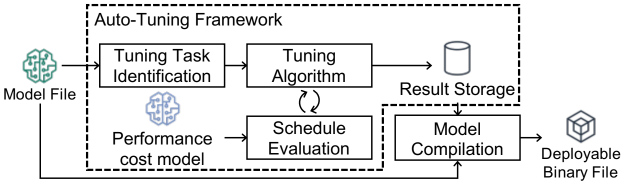

Figure 1: Auto-tuning frameworks.

workers is sufficient to establish a reliable task distribution mechanism.

Underneath the CLI, Lorien consists of five components. The *tuning task generator* generates a set of tuning tasks from a list of deep learning models to avoid redundant tuning while ensuring the operator coverage. The *distributed tuner* schedules tuning tasks to cloud instances or edge device farms to achieve high tuning throughput while guaranteeing the consistent tuning environment and tuning stability. The best tuning results as well as the complete tuning logs are maintained in a *database* and a file system, respectively. The *model builder* queries the best tuning results from the database to generate a deployable binary file for user-provided deep learning models. Optionally, we can use the *performance cost model trainer* with the complete tuning logs as the training data to train a performance cost model, which can be used to significantly facilitate future tuning processes. As we will illustrate in the next section, auto-tuning with the performance cost model is capable of identifying a high quality schedule in minutes in case the model builder receives a non-tuned task. Next, we introduce each component along with design choices and discussions.

3.1 Tuning Task Generator

The first step of auto-tuning is defining what to tune. This component aims to help users generate tuning tasks based on existing deep learning models for performance tuning. We summarize most possible model sources as follows, and design the tuning task generator accordingly.

Commonly used deep learning models. For this type of models, the performance of all workloads are crucial and should be optimized. Taking 29 commonly used CNN models from GluonCV model zoo [22] as examples, for auto-scheduling frameworks, we could construct 29 model-level tuning tasks. However, for template-based auto-tuning frameworks, we could additionally extract more than a thousand operator-level tuning tasks. It is worth noting that as deep learning continues thriving, the number of popular models and operators grow exponentially.

Deep learning models with variants. Users may need some variants based on commonly used deep learning models to fit different purposes, for example, different batch sizes for training or inference services. Another example is the convolution operators with different channel numbers or stride values to create a tuning space for neural architecture search (NAS) [8]. These variants could easily expand the number of tuning tasks by several times.

To generate tuning tasks from these sources, we design the generator in Figure 3. The generator accepts a set of model files and generates operator-, subgraph- and model-level tuning tasks. Since the definition of tuning tasks differs from each auto-tuning frameworks, how to parse deep learning model files and generate tuning tasks are transparent to framework specific dialects. Note that since the dialect is also in charge of generating a unique key for each tuning task, developers can also customize the logic of schedule sharing. For instance, the unique key of graph-level tasks can be a hash key from the serialized graph, so that two graphs with the same hash key can share the tuned schedules. Meanwhile, the unique key of operator-level tasks can be composed of the operator name, input tensor shapes, and attributes.

In addition, the tuning task identification dialect also accepts user-provided rules to mutate the tuning tasks for other applications, such as neural architecture search and dynamic batch training. Developers can customize rules with Python lambda expression to allow users to mutate specific values in tuning tasks. For example, the following rule mutates the batch size and channel number for 2D convolution tasks in operator-level. By applying this rule, the number of 2D convolution tasks will be increased by 7 × 3 = 21 times.

```
rules:
- task: conv2d_NCHWc.x86
  desc:
    batch: "lambda b: [1, 3, 4, 7, 8, 12, 16]"
    channel: "lambda c: [c, c * 2, c * 4]"
```

3.2 Distributed Tuner

To tune the tasks generated by the generator at scale, the Lorien tuner is designed in the master-worker pattern. Figure 4 presents the Lorien distributed tuner, which schedules all tasks to the workers and maintains the tuning state. The tuning master accepts a tuning task file in the YAML format, as well as the user-provided tuning configurations, such as the target platform and the number of tuning trials for each task. Then, the tuning master launches a task manager and tracks the progress. The task manager is able to schedule tasks to either cloud or edge workers based on the desired target device type of each task. Since the size of a tuning task and its tuning results are just a few KBs, the memory and network overhead of Lorien is tolerable compared to the standalone auto-tuning frameworks. Lorien leverages the corresponding auto-tuning framework dialect to perform tuning, which includes 1) the best schedule searching, and 2) schedule quality evaluation.

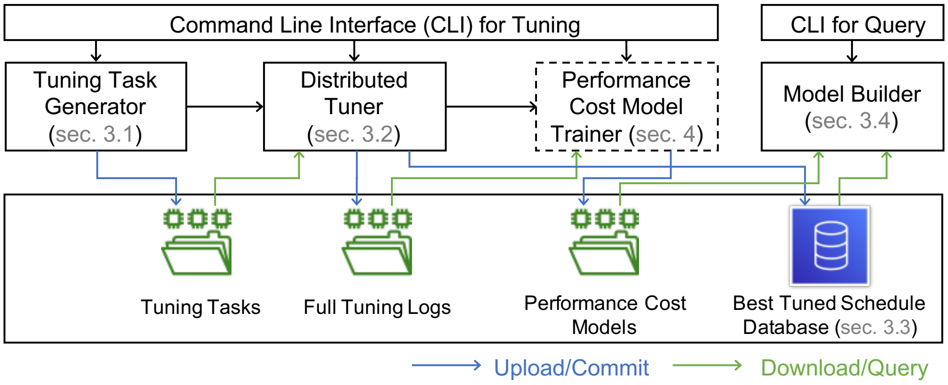
Command Line Interface (CLI) for Tuning
CLI for Query
Tuning Task Generator (sec. 3.1)
Distributed Tuner (sec. 3.2)
Performance Cost Model Trainer (sec. 4)
Model Builder (sec. 3.4)
Tuning Tasks
Full Tuning Logs
Performance Cost Models
Best Tuned Schedule Database (sec. 3.3)
Upload/Commit
Download/Query

Figure 2: Lorien system overview.

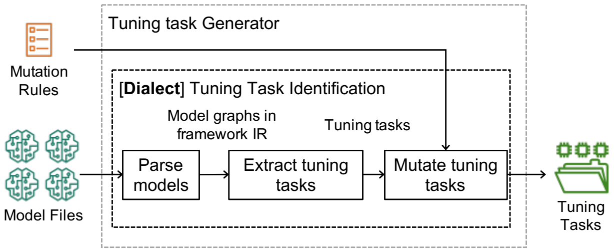

Figure 3: Tuning task generation.

Upon completion of the tuning, workers directly commit the best schedules to the database, and upload the complete tuning log to user-provided remote file system. Consequently, the tuning master could be lightweight as it does not have to aggregate the tuning results from workers.

On the other hand, although the failure of a lightweight master is unlikely, we further make the task manager stateless by tracing the state change of tasks to a trace file. Specifically, the master records job state changes, which overhead is less than a micro-second. In case the tuning master fails, another master can be relaunched with the trace file to recover all states and continue the tuning process. The recovery takes up to a few seconds so it will not affect the user experience.

Next, we present the mechanism of scheduling and managing tasks on cloud and edge in details.

Tuning tasks on cloud. Fortunately, modern public cloud services already have their own batch processing services that we can directly leverage, such as AWS batch [5], Google Cloud Dataflow [14], and Microsoft Azure Batch [30]. The batch processing services are in charge of scalability and reliability. They automatically launch required instances and schedule tasks on them. They will also resubmit tasks if the instance is terminated accidentally. As a result, Lorien simply submits jobs that use auto-tuning framework dialect to tune one task to the batch processing services, and regularly requests and analyzes the running logs from the service to update the tuning progress.

Tuning tasks on edge. On the other hand, many edge devices are not available on public cloud services and thus hard to be scaled out systematically. Developers usually need to build a device farm containing edge devices and several host machines, and manually construct a cluster system to manage these computation resources in terms of task scheduling and fault tolerance. On the other hand, Lorien task manager is capable of achieving the above requirements. Developers only need to launch a Lorien client on each host machine and let them connect to the tuning master. The client will register itself to be a worker, and it could start requesting tasks from the master and tune them on the connected edge devices. Different from most distributed systems that launch a server on each worker waiting for task allocations from the master, we choose to let the worker request tasks actively, because the device farms are usually behind a firewall while the tuning master could be a cloud instance or any machine with higher flexibility. In this case, the task manager allocates tasks to workers (device hosts) per their requests.

Another advantage of adopting a passive task manager is to support flexible workers. Users can register new workers or remove existing workers, which is common due to the unstable nature of the edge devices, without explicitly updating or even relaunching the tuning master. Since the connection protocol between the task manager and workers is RPC, the edge task manager, which is an RPC server, could keep tracking all connections. In other words, connections and disconnections will trigger corresponding event callbacks to update the worker list accordingly. It also ensures the system reliability, as the task manager is capable of recycling tasks from the disconnected workers immediately and re-allocating them to another available worker later on.

Note that like tuning on the cloud, how to tune a task on an edge device in a single machine is transparent to the

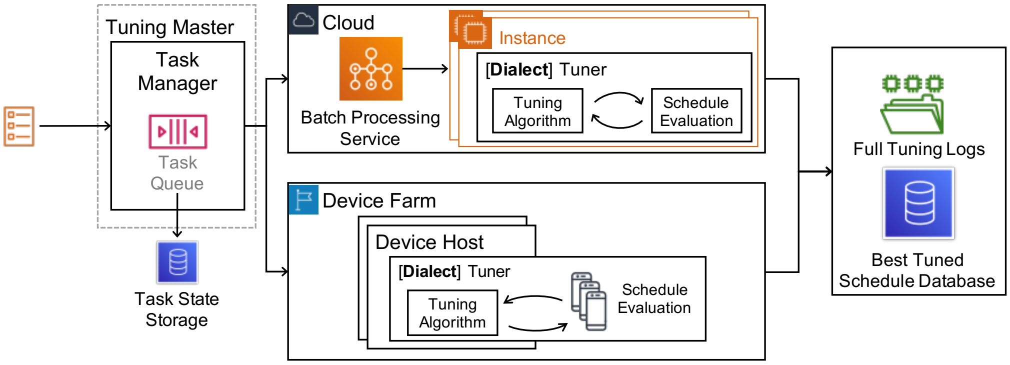

Figure 4: The distributed tuner that supports both cloud and edge platforms.

auto-tuning framework dialect in Lorien. For example, most deep learning compilers use resource-rich host machine to perform cross-compilation with the schedule being evaluated, and send the compiled binary to the edge device via remote procedure call (RPC) or similar protocol to measure their performance.

3.3 The Data Model

As mentioned in the previous subsection, Lorien distributed tuner leverages each worker to commit tuning results directly to the database to make the master lightweight. Accordingly, our tuning results and commit queries have the following characteristics. First, most queries are sent separately from different workers, so they can barely be batched. Second, one query will contain tuning results of one task. Third, tuning results of different tasks may have different attributes, because the argument list and schedule parameters of each task may vary. Fourth, tuning results of different tasks are mostly independent.

Based on the characteristics, we choose a scalable NoSQL database, Amazon DynamoDB [4], to manage the tuning results, as NoSQL databases are known to allow each object in a table to have flexible attributes. A DynamoDB table is composed of two components – attributes and indices. Attributes are the real data we intend to maintain. DynamoDB attributes can be in a simple type (e.g., `Int`, `String`) or a complex type (e.g., `List`, `Map`, `Item`). Consequently, a DynamoDB table item can also be hierarchical by putting another item in an attribute.

Indices determine how data will be stored, which significantly affect the query efficiency. DynamoDB key schema allows simple key (only one partition key) and composite key (one partition key with one sort key) to be table indices. Partition key determines how items will be physically partitioned and stored in hard disks. Sort key determines how items a in partition will be sorted. Accordingly, the table indices should be designed based on real use cases.

By summarizing auto-tuning frameworks for deep learning workloads, we design a unified data model that fits all frameworks. With the unified data model, developers can view the database as a black box, and the tuning results from different frameworks can be maintained together. Table 1 lists the attributes of a table item. In order to fast locate the table item to a certain tuning task, we need to query for 1) target platform, 2) tuning task key. As a result, we specify `Target` as the partition key, and create an extra attribute, `TaskKey` to be the sort key. The value of `TaskKey` is transparent to auto-tuning framework dialect when generating tasks. For example, for AutoTVM [12] or Halide auto-scheduler [2] that identify an operator or an entire model as a task, the task key can be composed of the operator or model name and its shapes and attributes; for Ansor [46] that identifies a subgraph as a task, the task key can be the serialized subgraph.

Besides, each item in the table includes a list of best schedules, as shown in Table 2. Note that since we use the binary type to store schedules, all forms of schedules (e.g., parameter values in template-based approaches and graph representations in auto-scheduling approaches) can be stored with the same data model. In addition to the schedule and its performance result, each best schedule item also includes 1) the auto-tuning framework configurations, such as the framework version as well as LLVM/CUDA versions, to make sure the result is reproducible, and 2) the full tuning log path in the file system.

When committing new tuning results of a task to the DynamoDB, we also check existing schedules (if any) tuned by the same framework with the same framework configuration and keep the better one. Meanwhile, the schedules tuned by different frameworks or configuration will be preserved for backward compatibility. The schedules with out-of-date framework configurations can be cleaned by a separate Lorien API, and this operation could be done offline during the regular maintenance period.

Table 1: The data model of DynamoDB table items.

Attribute	Type	Description
`TargetKind`	String	The kind of target platform for this task. e.g., `x86 CPU`, `ARM CPU`, `NVIDIA GPU`
`Target`	String	The full target with detail attributes. e.g., `Intel Xeon Platinum 8124M`
`TaskKey`	String	A unique key of the task. This is defined by auto-tuning framework dialect when generating tasks.
`TaskName`	String	The tuning task name, which could be the name of an operator or a hash code of a graph.
`Args`	List	The task arguments or attributes .
`BestSchedules`	List	The best schedules (See Table 2.)

Table 2: The data model of best schedules.

Attribute	Type	Description
`Latency`	Number	Measured latency.
`Thrpt`	Number	GFLOP/second.
`Config`	Map	Framework build configure.
`Schedule`	Byte	Serialized schedule.
`LogPath`	String	The full tuning log path.

Table 3: Examples of auto-tuning framework configuration distance computation.

Config	Framework Version	LLVM Version	CUDA Version	Bit Array	Distance
Desired	0.6.1	8.0	10.2	000	0
A	0.6	8.0	10.2	100	4
B	0.6.1	9.0	10.1	011	3

3.4 Deep Learning Model Builder

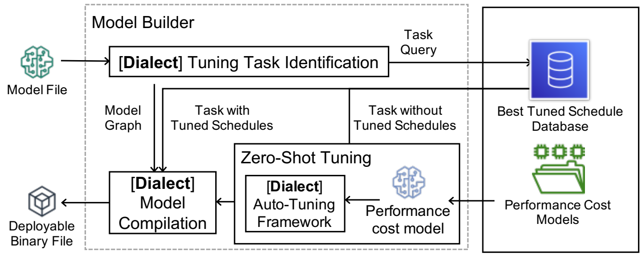
Model Builder
Task Query
[Dialect] Tuning Task Identification
Model File
Model Graph
Task with Tuned Schedules
Task without Tuned Schedules
Best Tuned Schedule Database
Zero-Shot Tuning
[Dialect] Model Compilation
[Dialect] Auto-Tuning Framework
Performance cost model
Performance Cost Models
Deployable Binary File

Figure 5: Deep learning model compilation.

With the schedule database committed by Lorien tuning workers, users can now query for the best schedule when compiling a deep learning model. The execution flow is shown in Figure 5. We first follow the same path as the tuning task generator to parse the input model and extract its tuning tasks. On the other hand, instead of tuning the tasks, this time we only use their targets and task keys to perform batch querying to the database. After that, we will receive a list of best schedules for each task that has been tuned in advance.

In order to make sure the queried schedule fits the user environment (e.g., framework version or dependent toolkit version), we only select the schedule with exactly the same `Config`. However, different framework configurations might be acceptable sometimes. For example, the schedule based on the framework built with LLVM 8.0 may also fit to the same framework built with LLVM 9.0. To deal with the case that users may want Lorien to return the schedule from the similar configurations, we allow users to specify a list of acceptable configuration fields, and return the desired schedules with the closest framework configurations. Table 3 provides a simple example of computing configuration distances. The `Desired` configuration is the one we desired for, and we want to determine whether configuration `A` or `B` is closer to the target.

We use a bit array to define the distance between configurations. By setting the bit array of the `Desired` configuration to zeros, we set a particular bit of a candidate configuration to 1 if the corresponding value is different from the `Desired`. By prioritizing the fields from left to right, we can simply compare the resulting bit arrays of candidates to get the one with smallest discrepancy, which is `B` configuration in this example.

Intuitively, in case the workload has no schedule in the database, or all schedules are generated with incompatible environment configurations, we can "borrow" schedules of the workload from other platforms (e.g., apply a schedule from NVIDIA T4 to NVIDIA V100). However, as we will evaluate in subsection 5.1, one-for-all schedule does not exist, meaning that the efficient schedule on a platform is usually not efficient on another. Alternatively, it is promising to perform zero-shot tuning in minutes with the help from the performance cost model trained by the tuning data, which will be presented in the next section.

4 PERFORMANCE COST MODEL AND ZERO-SHOT TUNING

As mentioned in subsection 3.4, it is inevitable that some queried workloads are not covered in the database due to the diversity of deep learning architectures. For example, the 2D convolution operator has 13 free variables, such as input tensor shapes, stride, padding, dilation, etc., and the number of combinations of all variables could be in billions, which is impractical for a database to fully cover. Consequently, existing auto-tuning frameworks [2, 12, 46] leverage performance cost models, which predict the performance of the given schedule, to reduce the time-consuming on-device measurements. Formally, for features $\vec{x}$ extracted from a given schedule, the performance cost model predicts a score y based on $\vec{x}$, in which a higher number indicates a schedule with better performance in terms of latency or throughput.

Ideally, when the performance cost model is capable of accurately ranking the quality of schedule candidates, we can obtain the optimal schedule by purely leveraging the performance cost model without any on-device measurement. We call this approach *zero-shot tuning*, because the cost model is fixed and is used as the target for black-box optimization. On the other hand, there are two challenges for the performance cost models adopted by existing auto-tuning frameworks to support zero-shot tuning on various platforms.

Challenge 1: Portability. Performance cost models proposed by existing auto-tuning frameworks leverage hardware features or machine learning models. Although effective hardware features make their models achieve sufficient accuracy with a large-scale dataset, the heavily engineered hardware features cannot be easily ported to another hardware platform.

Challenge 2: Model fine-tuning. Existing auto-tuning frameworks often fine-tune their cost models continuously along with newly measured data during the tuning process. In this way, on-device measurement is still a major bottleneck for a long tuning time, because on-device measurement involves schedule compilation, data transfer, and on-device execution. In fact, to achieve the best performance, existing auto-tuning frameworks often require at least thousands of on-device evaluations.

To address the challenges, Lorien shipped with a performance cost model trainer, as shown in Figure 6, that leverages *an AutoML solution* to automatically train a suitable performance cost model for a certain hardware platform with tuning history.

Lorien aims to leverage AutoML to find the best model architecture for the target hardware device, and train the model on *high-level* features extracted from an auto-tuning framework specific dialect, which simply parses and formalizes measured schedules in framework specific representation. The generated features can be hardware independent, such as the schedule parameters like tile size and loop unrolling in a template-based approach (e.g., AutoTVM [12]), or the sequence of scheduling actions in auto-scheduling approach (e.g., Halide auto-scheduler [2] and Ansor [46]). In this way, the efforts of feature/model engineering for new operator and hardware devices can be significantly reduced (challenge 1).

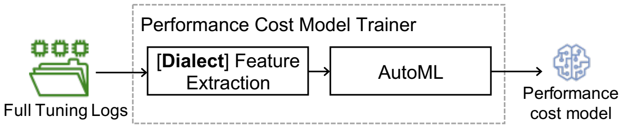

Figure 6: Performance cost model trainer. The dialect is used to extract *high-level* features from auto-tuning framework specific schedule representations.

Although building an accurate and portable performance cost model from high-level features is considered challenging because the model has no visibility to the low-level hardware-specific features and can only figure out these intrinsic features by looking at the data, our evaluation in subsection 5.2 shows that this is possible if the performance tuning dataset used to train the model is at *large scale*. It implies that this approach is tightly-coupled with efficiently tuning massive workloads. This shares the same rationale as the breakthroughs in computer vision [15] and natural language processing [34] that are realized by scaling up the datasets. In addition, with AutoML, the training phase is transparent to the user and it works well for different types of hardware and workloads. In fact, we will show that the model trained by AutoML significantly outperforms the baseline models, including CatBoost regression and ranking models [33] and neural network.

For a new workload that is missing in the database, Lorien will first randomly sample thousands of scheduling candidates and rank them with the cost model, and it only evaluates the top few candidates (e.g., 8 or fewer) on device. The process does not require fine-tuning and will be significantly faster than tuning from scratch, which will involve iterative on-device evaluation and cost model update. Since the AutoML-based model is accurate, this algorithm is capable of producing a schedule that performs similarly to the optimal schedule stored in the database. As a result, combining the AutoML-based cost model and zero-shot tuning enables Lorien to deliver a decent schedule of unseen workloads in minutes (challenge 2).

5 EVALUATION AND ANALYSIS

Lorien currently supports two dialects – AutoTVM [12] and Ansor [46]. We have used Lorien to preform massive tuning tasks to collect schedules on 29 widely-used models in Gluon CV model zoo [22] for both dialects. This gives us a list of representative and computationally intensive deep learning operators and workloads.

We take the AutoTVM tuning results in this section to first analyze and discuss the insights in subsection 5.1, followed by an evaluation of the performance cost model and zero-shot tuning algorithm in subsection 5.2. We note that the tuning results of Ansor derive to the same insights, and the model speedup is not the major contribution of this paper, so we only focus on AutoTVM results in this section for concise. From these models, AutoTVM extracted 567, 633, and 1,148 operator-level tuning tasks for x86 CPU, ARM CPU, and NVIDIA GPU, respectively. Note that since an operator (e.g., 2D convolution) may have more than one implementations (e.g., direct and Winograd algorithm [44] implementations), the number of extracted tasks for different hardware platforms could be different.

Table 4: Covered hardware platforms.

Platform	Backend	Target Device
Amazon EC2 C4	x86	Intel Xeon E5-2666 v3
Amazon EC2 C5	x86	Intel Xeon Platinum 8124M
Amazon EC2 G4dn	CUDA	NVIDIA T4 Tensor Core
Amazon EC2 P3	CUDA	NVIDIA Tesla V100
Amazon EC2 M6g	ARM	AWS Graviton2
Raspberry Pi 4B	ARM	Cortex-A72 (ARM v8)

We tuned all above tasks on several hardware platforms, including cloud and edge devices, as shown in Table 4. Each tuning task was tuned by AutoTVM for at most 5,000 trials, meaning that each task will have at most 5,000 schedules. With AWS batch for the cloud platforms and a self-maintained device farm for the edge devices, we spent less than 2 days to finish all the tuning.

5.1 Tuning Result Analysis

With a large amount of tuning results by Lorien, this subsection analyzes the tuning results on several platforms. Such analysis has not been done before due to the lack of a comprehensive dataset generated by a scale-out system. Specifically, we seek to answer the following questions:

- What is the best performance we can achieve, and how many trials are actually required?
- Can we directly apply the best schedule of one platform to another so that we can tune fewer tasks?

Table 5: Average iteration numbers to find the schedule that achieves 90% of the best achieved performance.

Platform	Intel E5	Intel Platinum	AWS Graviton 2
Avg. Iter No.	628	591	1253
Platform	ARM Cortex-A72	NVIDIA T4	NVIDIA V100
Avg. Iter No.	656	1136	1101

Figure 7 presents the speedup of a number of deep learning models from Gluon CV model zoo tuned by AutoTVM with Lorien. Note that the speedup on x86 platforms shown in the figure considers graph tuning [29], which further improves the end-to-end network performance by minimizing the data layout transformation overhead between operators. The baseline is either the hard-coded default schedules in TVM, or the available schedules on TopHub [38]. As can be seen in the figure, ResNet-18 achieves moderated speedup on all platforms (1.2×). This is because ResNet was the most widely used model, so most auto-tuning frameworks, including AutoTVM, guarantee the quality of default schedules of ResNet covered operators. In contrast, we can observe significant speedup (8.12×) for a less popular model such as SqueezeNet. This implies that Lorien is able to play an important role for new emerging deep learning models.

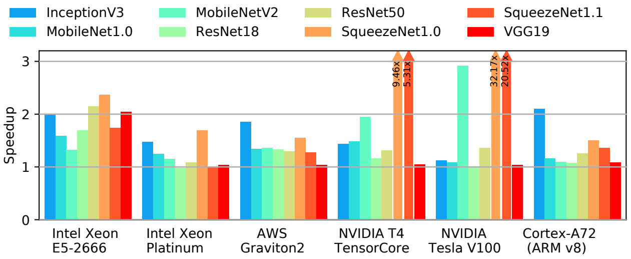

Figure 7: Speedups achieved by the best schedules tuned by AutoTVM and Lorien over the default schedules in TVM and TopHub [38].

In addition, the achieved speedups on GPUs (5.28×) are higher than on CPUs (1.45×) in average. This attributes to the fact that GPU schedules have much larger tuning space, which makes it harder for schedule template designers to figure out an effective default schedule for all operators.

Meanwhile, with the tuning data of plentiful tuned operators on multiple platforms, Table 5 summarizes the average iteration counts that first find the schedule achieving 90% performance over the best schedule. Note that the latency of an operator is in the scale of a few μs, so 90% of the best performance is usually sufficient for most use cases. As can be seen, although we use Lorien to tune each operator for at most 5,000 trials, which usually covers more than 90% of the tuning space on CPU and about 0.01% on GPU, much less trial numbers could be sufficient to serve most use cases. Consequently, tuning time could be saved for users that are eager to have a model with decent performance, but Lorien can still perform several thousands of trials to achieve the peak performance.

Table 6: Rank shifting analysis when porting the best 32 schedules from one platform to another. "Ratio" in the last column means the ratio of schedules that achieve the same or better performance on the ported platform.

Original Platform	Target Platform	Ratio
Intel E5	Intel Platinum	5.34%
Intel Platinum	Intel E5	5.42%
ARM Cortex-A72	AWS Graviton 2	4.18%
AWS Graviton 2	ARM Cortex-A72	4.22%
NVIDIA T4	NVIDIA V100	19.84%
NVIDIA V100	NVIDIA T4	20.42%

Next, we evaluate the schedule portability. We select the best 32 schedules of each workload on one platform and measure their performance on another platform to see if they remain the best. Table 6 shows the results, where rank improvement indicates the ratio of schedules achieving the same or better rank on the target platform. As shown in the table, the ranks of top schedules on one platform are mostly worse than on another, and only about 6% and 20% of the top 32 schedules can achieve the same or better rank on another CPU and GPU platforms, respectively.

Based on the analysis result, we conclude that *one-for-all* schedules basically do not exist, and the most practical approach to achieve a decent performance on a new hardware platform is tuning desired workloads on the platform directly. This further motivates the Lorien to provide a scalable and reliable tuning mechanism as well as schedule management.

5.2 Performance Cost Model Evaluation

In this section, we evaluate the accuracy of the AutoML-based performance cost model trained for each hardware platform. As Table 6 illustrated, high quality schedules for a hardware platform usually perform poor on another, which implies that it is hard for the performance cost model trained for one hardware to be transferred to another. However, we will show that platform-specific performance cost model is still effective and useful for zero-shot tuning algorithm proposed in section 4 to rapidly deliver a decent schedule for unseen workloads on the same platform. Based on the experiment results, our main findings are

- We can obtain an accurate AutoML-based performance cost model by training on a large-scale dataset for a certain device. The dataset scale plays an important role in the performance.
- The AutoML-based solution significantly outperforms baselines like CatBoost regression / ranking or neural network and works well on datasets collected from different hardware devices.
- Zero-shot tuning with the AutoML model is able to identify a schedule that is comparable to the best schedule searched from scratch. This reduces the tuning time of unseen workloads from hours to around one minute.

We implement the AutoML-based solution via AutoGluon [16]. The default configuration of AutoGluon trains 11 different models that belong to different categories such as boosting tree, random forest, and neural network, and uses their weighted ensemble. However, during our experiments, we observed that the weighted ensemble model will induce a high inference latency. To improve the inference speed of the cost model, we applied a simple pruning strategy on top of AutoGluon (hereinafter referred to as "*AutoGluon + Pruning*" below) by keeping a single model with the highest validation accuracy out of the 11 models. By analyzing the validation accuracy of 11 models, we observed that neural network (NN) models generally outperform tree models on GPUs while tree models outperform NN models in other hardware. This again proves that a model crafted by humans for a device may not work well on another, and AutoML could be a unified solution to free human efforts while covering most devices.

Our baselines are three representative cost models that are widely used machine learning models for predicting the throughput scores [11, 35]: 1) a boosting-tree-based regression model that minimizes the Root-Mean-Square Error (RMSE) between the predictions and the ground-truth throughputs, 2) a boosting-tree-based model that minimizes a list-wise ranking objective [21], and 3) a neural network (NN) model that minimizes Mean-Squared Error (MSE). We adopted the regression and ranking models implemented in CatBoost [33] as the boosting-tree-based models. For the neural network based model, we stack three MLP layers with batch normalization, dropout, and leaky ReLU activation. All models are trained with the data Lorien collected.

Each dataset contains the performance data of one operator on a particular hardware platform. Each sample in the dataset is a pair of scheduling features and its corresponding throughput. In our experiments, we use 15 datasets in total. Each dataset contains schedules from multiple workloads

Table 7: Comparison of different performance models. "NDCG@k" means the NDCG score of the top-k schedules. The AutoML-based solution performs significantly better than the other models. "AutoGluon + Pruning" means to apply the simple pruning strategy on top of AutoGluon.

Metric	NDCG@2 ↑	NDCG@8 ↑
CatBoost Regression	0.8682	0.8735
CatBoost Ranking	0.7775	0.7899
NN	0.8387	0.8520
AutoGluon	**0.9220**	**0.9324**
AutoGluon + Pruning ★	0.9216	0.9301

(e.g., the Conv2D CUDA dataset has more than 2M schedules from 543 workloads in terms of input/output shapes, kernel/padding sizes, data types and layouts). To make sure the trained model works well with new workloads on the same platform, we split the training and testing sets based on workloads and ensure that the workloads in the testing set do not appear in the training set. We randomly sample 10% of the workloads as the testing set, and the rest are kept as the training set.

In the zero-shot tuning setting, we will 1) randomly sample N schedules, 2) estimate their performance with the cost model, and 3) keep the top-K schedules with the highest score and evaluate their throughputs on device. Since only the predicted relative ordering among schedules matters in this algorithm, we evaluate cost models with the Normalized Discounted Cumulative Gain (NDCG) scores [24] at different ranks. Assume that there are N predicted scores $\{y_1, \cdots, y_N\}$ and N throughputs $\{\text{thrpt}_1, \cdots, \text{thrpt}_N\}$, to calculate NDCG_k, we sort the predicted scores in descending order and calculate the DCG score defined as $\text{DCG}_k = \sum_{i=1}^{k} \frac{2^{\text{thrpt}_{\text{pos}_i}} - 1}{\log(1+i)}$, in which pos_i means the position of the i-th largest schedule. NDCG_k is then defined as $\text{NDCG}_k = \frac{\text{DCG}_k}{\text{IDCG}_k}$, in which IDCG_k is the ideal and also maximal DCG score obtained by the perfect ordering. In short, NDCG at rank k, or NDCG_k, measures the quality of the top-k predictions of the cost model. If NDCG_k is 1, it indicates that the top-k predicted scores have exactly the same order as the ground-truth. Since NDCG_k measures the quality of the top-k predictions, it is a good proxy of the performance of the top-k candidates searched via zero-shot tuning.

The main comparison results are shown in Table 7. We train each model independently on 15 datasets and report the average scores. We can find that AutoGluon significantly outperforms the baseline models In addition, the "AutoGluon + Pruning" model has almost the same performance as the AutoGluon model, but is more than 10 times faster. Thus, we choose to use "AutoGluon + Pruning" in the rest of this evaluation.

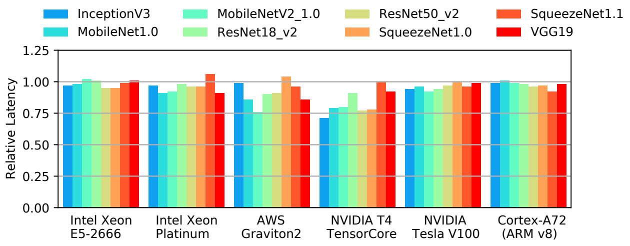

Figure 8: Speedups achieved by the best schedules tuned by zero-shot tuning with the performance cost model over the best schedules in Figure 7.

We also conducted end-to-end zero-shot tuning experiment with our performance cost model on the GluonCV models. In this experiment, we make sure the workloads in the GluonCV model being tuned do not appear in the training set. For each task, we randomly search for 2, 000 trials and only measure the top-8 schedules on device predicted by the cost model. We report the end-to-end speedup of deep learning models compiled with schedules obtained via zero-shot tuning over the best schedules stored in the database. The results are shown in Figure 8. In most cases, zero-shot tuning is able to achieve comparable performance to the best schedule. In some cases, it even outperforms the best schedule in the Lorien database, which suggests that the performance cost model generalizes well and can lead to even better schedules. In addition, we also evaluated the average tuning time saved by the zero-shot tuning. On average, by zero-shot tuning with the performance cost model, the tuning time for a new workload reduces from 25 minutes to 53 seconds on x86 platforms (Intel Xeon CPUs); 3.3 hours to 28 seconds on GPU platforms (NVIDIA V100 and T4); 3.8 hours to 77 seconds on edge platforms (ARM v8 Cortex-A72).

Moreover, to understand the impact of dataset scale on the NDCG score, we sub-sample the training data to 70%, 50%, and 30% to train the "AutoGluon + Pruning" model. The NDCG@2 and NDCG@8 scores on the test set are shown in Figure 9. The result clearly shows that the scale of the training set is essential to achieve acceptable accuracy. This experiment shows that the large-scale dataset provided by Lorien can help build more effective performance cost models to achieve good performance in zero-shot tuning.

Finally, we conducted an ablation study to understand the impact of two important hyper-parameters of zero-shot tuning: 1) the number of schedule candidates that are measured on-device, and 2) the number of tuning trials with the cost model. To understand the impact of the number of on-device measured schedules, we fix the number of tuning trials to be

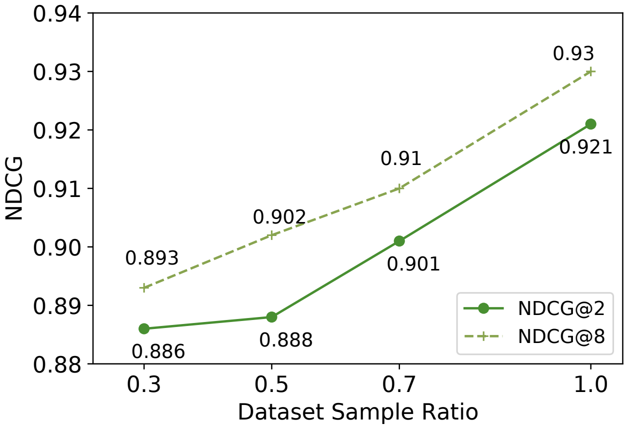

Figure 9: The impact of dataset scale on the NDCG score. We subsample the training data with different ratios and report the average NDCG@2 and NDCG@8 scores.

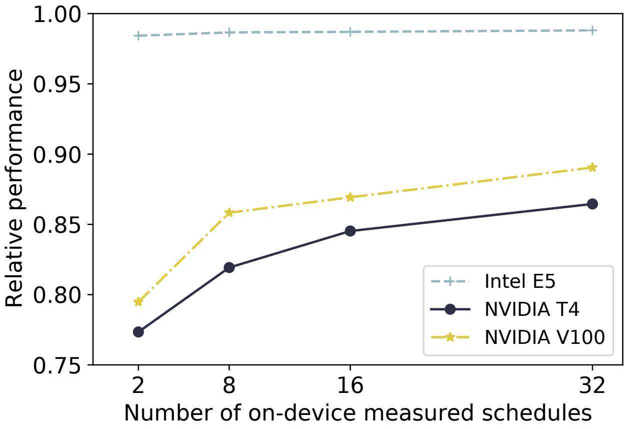

Figure 10: The impact of on-device measured schedules performed at the end of zero-shot tuning. We report the relative performance with respect to the schedules stored in the DB.

1,000 and run the end-to-end tuning with different numbers of schedule candidates that are measured on-device. The results are shown in Figure 10. We can find that the performance improvement from measuring 8 top schedules to 16 and 32 is not obvious. This means that the cost model is accurate enough to rank the best schedule.

To understand the impact of tuning trials, we fix the number of on-device measured schedules to be 8 and run end-to-end tuning with different numbers of tuning trials. The

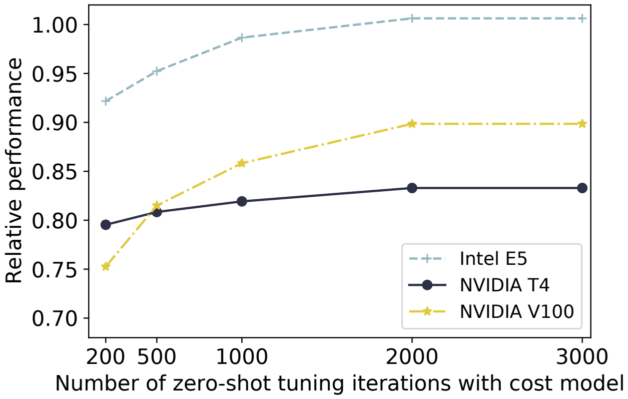

Figure 11: The impact of iterations performed by zero-shot tuning. We report the relative performance with respect to the schedules stored in the DB.

results are shown in Figure 11. We observe that better performance can be obtained by increasing the number of tuning trials, which converges at about 2,000 trials. It means that randomly sampling 2,000 candidates and ranking them with the cost model in a few seconds is sufficient to deliver efficient schedules that achieve 80+% performance against the auto-tuning frameworks, which makes zero-shot tuning practical.

6 RELATED WORK

In this section, we discuss state-of-the-art auto-tuning frameworks for deep learning workloads. Then, we introduce related performance cost models used in these frameworks that help facilitate the tuning process.

Deep Learning Tuning Frameworks for Clusters Some existing deep learning tuning frameworks are designed for efficiently training deep learning models with hyper-parameter tuning on cloud. For example, Amazon SageMaker [28], Google Vizier [19], Auto-Keras [25] are the cloud-based frameworks for hyper-parameter tuning. On the top of that, PipeTune [37] also considers system parameters to further reduce the model training time while achieving the same accuracy. Their framework designs share a degree of similarity of Lorien; while Lorien focuses on the model inference performance and faces different challenges.

Deep Learning Kernel Auto-Tuning Frameworks We categorize auto-tuning frameworks to two categories, based on their tuning task granularity. The first category [2, 39, 42] schedules an entire deep learning model together by treating the entire model as a single graph. Halide auto-scheduling [2, 39] uses loop transformation primitives to construct a tuning space, and leverages tree-based beam search with heuristic pruning strategies to find the best schedule.

Tensor Comprehensions (TC) [42] is a framework built on the polyhedral model and targets GPU. It leverages evolutionary search to explore the tiling sizes and different strategies of polyhedral transformation. Since these frameworks schedule each operator in a deep learning model based on the scheduling result of the previous operator, the tuned schedule is tightly-coupled to the deep learning model. When integrating into Lorien, the tuning task generator dialect simply generates one task for an entire model with the serialized model graph as the unique task key.

The second category [12, 46, 47] partitions the deep learning model graph to several independent subgraphs so that they can be tuned in parallel. AutoTVM [12] and FlexTensor [47] partition the model graph by operators (e.g., `Conv2D`); while Ansor [46] partitions the model graph by heuristic rules (e.g., `Conv2D-add-ReLU`). The tuning task generator dialect in Lorien of these frameworks could generate several tuning tasks with task key composited by operator name or serialized subgraph.

In addition, since almost all frameworks in both categories have corresponding user interfaces to accept pretrained performance cost model, it is straightforward for Lorien to train and plug in the cost model for them.

Performance Cost Models

Most auto-tuning frameworks [2, 12, 18, 39, 46] incorporate machine learning cost models in the performance tuning. To accurate the performance score, AutoTVM [12], Ansor [46], and Opevo [18] use a cost model with XGBoost [9]; Halide auto-scheduling [2, 39] builds regression models with hardware features. In addition, some existing work [6, 26, 40] attempt to leverage high accurate cost models to reduce the impact of search algorithm on final performance. [6] proposes a Recurrent and Recursive Neural Network model for Tiramisu [7] domain specific language used as a polyhedral compiler for dense and sparse deep learning. [26] adopts Graph Neural Network (GNN) to predict the program runtime on Google Tensor Process Units (TPUs). [40] formulates the tuning process as a deterministic Markov Decision Process (MDP), and solves it by learning an approximation of the value function. Due to the effective hardware dependent features crafted with heavy feature engineering, these models can be well-trained by a dataset with only a few thousands of schedules. However, hardware-specific features usually cannot be easily ported to new hardware, so this approach cannot keep pace with the rapid development of a new hardware.

Apart from ML-based cost models, there is a stream of works that exploit the analytical models to estimate the performance [27, 41, 42, 47]. Although analytical model does not require any training data and can further reduce the hardware requirements, analytical models are also restrained to a certain hardware platform or operator, which makes them less useful to be deployed in general deep learning compilers.

7 CONCLUSIONS AND FUTURE WORK

This paper presents Lorien, a unified and extensible infrastructure for delivering efficient deep learning workloads upon requests. Lorien allows auto-tuning deep learning frameworks to be easily plugged in as dialects, and supports large scale tuning on both cloud and edge platforms. The tuning results are managed in a NoSQL database with a unified data model that fits all auto-tuning frameworks. While the best schedules managed in the database can be used to compile deep learning models to achieve high performance, the tuning logs managed in a file system can also 1) enable more comprehensive performance analysis on different platforms, and 2) help train a performance cost model with an AutoML solution. The evaluation result shows that the trained performance cost model is accurate enough to avoid fine-tuning during the workload tuning process, which enables zero-shot tuning and guarantees to deliver efficient schedules in seconds or minutes.

As we are expecting a continuous effort from both industry and academia to improve auto-tuning in the future, Lorien is expected to share the benefits they bring. In addition, we envision Lorien could also bring the following future work.

Operator-level performance benchmarking. Many deep learning accelerators have been developed to serve different scenarios. To know the model performance on certain hardware platforms, MLPerf [36] benchmark suite is proposed. While MLPerf focuses on end-to-end performance, Lorien database could also cover the schedule and performance at operator-level, so that hardware vendors could easily catch up with the performance on state-of-the-art accelerators to set up the goal of their devices.

Latency/memory optimization in NAS. The recent trend of neural architecture search (NAS) [8, 43] is to find architectures that have both good performance and low latency/memory cost on the downstream hardware. Lorien can help these hardware-aware NAS algorithms build a more fine-grained (operator-level) and accurate performance model.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[3] Amazon. 2006. Elastic Compute Cloud. https://aws.amazon.com/ec2/

[4] Amazon. 2012. DynamoDB. https://aws.amazon.com/dynamodb/

[5] Amazon. 2016. AWS Batch. https://aws.amazon.com/batch/

[6] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. *Proceedings of Machine Learning and Systems* 3 (2021).

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[8] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*.

[9] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, 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.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*. 3389–3400.

[13] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[14] Google Cloud. 2014. Dataflow. https://cloud.google.com/dataflow

[15] Jia Deng. 2012. *Large scale visual recognition.* Technical Report. PRINCETON UNIV NJ DEPT OF COMPUTER SCIENCE.

[16] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505* (2020).

[17] Facebook. 2016. torchvision. https://github.com/pytorch/vision

[18] Xiaotian Gao, Cui Wei, Lintao Zhang, and Mao Yang. 2020. OpEvo: An Evolutionary Method for Tensor Operator Optimization. *arXiv preprint arXiv:2006.05664* (2020).

[19] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 1487–1495.

[20] Google. 2019. TensorFlow Hub. https://tfhub.dev/

[21] Andrey Gulin, Igor Kuralenok, and Dimitry Pavlov. 2011. Winning the transfer learning track of yahoo!'s learning to rank challenge with yetirank. In *Proceedings of the Learning to Rank Challenge*. 63–76.

[22] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, et al. 2020. GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing. *Journal of Machine Learning Research* 21, 23 (2020), 1–7.

[23] Intel. 2020. Intel® Math Kernel Library for Deep Learning Networks. https://github.com/oneapi-src/oneDNN

[24] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[25] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1946–1956.

[26] Samuel J Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, and Mike Burrows. 2020. A Learned Performance Model for the Tensor Processing Unit. *arXiv preprint arXiv:2008.01040* (2020).

[27] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P Sadayappan. 2019. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[28] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. 2020. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 731–737.

[29] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. 2019. Optimizing CNN Model Inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*. 1025–1040.

[30] Microsoft. 2017. Azure Batch. https://azure.microsoft.com/en-us/services/batch/

[31] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 83.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8024–8035.

[33] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: Unbiased boosting with categorical features. In *NeurIPS*. 6638–6648.

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[35] 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. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[36] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf Inference Benchmark. arXiv:1911.02549 [cs.LG]

[37] Isabelly Rocha, Nathaniel Morris, Lydia Y Chen, Pascal Felber, Robert Birke, and Valerio Schiavoni. 2020. PipeTune: Pipeline Parallelism of Hyper and System Parameters Tuning for Deep Learning Clusters. In *Proceedings of the 21st International Middleware Conference*. 89–104.

[38] SAMPL Research Group. 2018. TVM-Distro. https://github.com/uwsampl/tophub

[39] Savvas Sioutas, Sander Stuijk, Twan Basten, Henk Corporaal, and Lou Somers. 2020. Schedule Synthesis for Halide Pipelines on GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 3 (2020), 1–25.

[40] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value Learning for Throughput Optimization of Deep Learning Workloads. *Proceedings of Machine Learning and Systems* 3 (2021).

[41] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Gagandeep Goyal, Ramakrishna Upadrasta, and Bharat Kaul. 2020. PolyDL: Polyhedral Optimizations for Creation of High Performance DL primitives. *arXiv preprint arXiv:2006.02230* (2020).

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[43] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. 2020. Hat: Hardware-aware transformers for efficient natural language processing. In *ACL*.

[44] Shmuel Winograd. 1980. *Arithmetic complexity of computations*. Vol. 33. Siam.

[45] XLA Team. 2017. XLA - TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[46] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. *arXiv preprint arXiv:2006.06762* (2020).

[47] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 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.