Trimmer: Cost-Efficient Deep Learning Auto-tuning for Cloud Datacenters

Paper ID: 40

***Abstract*—Cloud datacenters capable of provisioning high performance Machine Learning-as-a-Service (MLaaS) at reduced resource cost is achieved via auto-tuning: automated tensor program optimization of Deep Learning models to minimize inference latency within a hardware device. However given the extensive heterogeneity of Deep Learning models, libraries, and hardware devices, performing auto-tuning within Cloud datacenters incurs a significant time, compute resource, and energy cost of which state-of-the-art auto-tuning is not designed to mitigate. In this paper we propose Trimmer, a high performance and cost-efficient Deep Learning auto-tuning framework for Cloud datacenters. Trimmer maximizes DL model performance and tensor program cost-efficiency by preempting tensor program implementations exhibiting poor optimization improvement; and applying an ML-based filtering method to replace expensive low performing tensor programs to provide greater likelihood of selecting low latency tensor programs. Through an empirical study exploring the cost of DL model optimization techniques, our analysis indicates that 26–43% of total energy is expended on measuring tensor program implementations that do not positively contribute towards auto-tuning. Experiment results show that Trimmer achieves high auto-tuning cost-efficiency across different DL models, and reduces auto-tuning energy use by 21.8–40.9% for Cloud clusters whilst achieving DL model latency equivalent to state-of-the-art techniques.**

***Index Terms*—Deep Learning, Cloud datacenter, MLaaS, Machine Learning systems, Energy, Sustainable AI**

I. Introduction

Deep Learning (DL) has become increasingly important across industry and academia, with numerous DL models created to perform sophisticated Computer Vision and Natural Language Processing [1]. Creation of new DL model architectures combined with growing user demand has resulted in the formation of Cloud datacenters containing specialized hardware devices (GPUs, FPGAs, etc.) dedicated to provisioning Machine Learning-as-a-Service (MLaaS) [2]. There is a major impetus for providers to ensure that such Cloud datacenters are capable of provisioning, as well as creating, DL models with high accuracy and low latency inference to achieve high system throughput and cost-efficiency in terms of compute resource and energy consumption. An effective means to attain this goal is to optimize the individual computational components of DL models - *tensor programs* - towards specific target-device characteristics, spanning cache/memory access patterns, thread processing, and hardware-intrinsic functions [3].

Performing tensor program optimization is a complex and time-consuming task, which has in turn resulted in the creation of *auto-tuning*: automated DL model optimization of tensor program implementations towards a target-device. Facilitated by DL compilers such as TVM [3] and Halide [4], auto-tuning minimizes tensor program latency via an iterative exploration of a large tensor program parameter space, and necessitates tens of thousands of costly iterative measurements of candidate implementations per target-device. Auto-tuning is frequently deployed within DL-focused Cloud datacenters [5]–[7].

Whilst numerous auto-tuning techniques have been proposed to accelerate the candidate search of tensor programs [4], [8]–[11], the auto-tuning process requires considerable time and compute resource cost to complete, whereby even a small DL model must occupy an isolated target-device for hours to yield reasonable performance improvements [10]. This becomes a considerable issue in the context of Cloud datacenters which must provision high performance and cost-efficient MLaaS (including user defined auto-tuning [5]) for a large user base each with unique DL model configurations and hardware device constraints. This translates into monetary loss for Cloud providers stemming from longer auto-tuning duration and waiting times, lower MLaaS throughput and availability due to auto-tuners requiring exclusive access to a target-device for extended time periods, and incurs higher energy consumption, representing a barrier towards creating environmentally sustainable AI systems [12].

In this paper we present Trimmer: A high performance and cost-efficient DL auto-tuning framework that reduces the total time and energy cost required to perform tensor program optimization for MLaaS deployed within Cloud datacenters. Driven by an empirical study of DL optimization performance and energy cost, Trimmer proposes a ML-based candidate filtering to reduce the number of hardware measurements of candidates identified as likely to fail, and re-sample the candidate search space to replace expensive long-running *cold* candidates with faster, *hot* candidates increasing the likelihood of finding faster tensor programs. Trimmer is capable of performing hardware measurements of tensor program implementations as batches in parallel that are periodically compiled to measure improvements to model inference latency, rather than probing the search space serially. Such an approach allows Trimmer to prioritize or discard tensor program optimizations across a cluster of machines based on relative performance speed-up across multiple DL models whilst achieving equivalent inference latency compared to state of the art.

The core contributions of our work are as follows:

- We analyse the heterogeneous energy and performance characteristics of DL model optimization frequently used for MLaaS, and show that cold candidates generate 26–43% of energy waste within the auto-tuning process;
- We demonstrate that by extracting the intermediate layer of the neural network, we can query similar tensor program candidates by using cosine similarity measure to filter poor performing candidates;
- We show via experimentation that Trimmer provides cost-efficient auto-tuning, and within cloud DL clusters reduces the system energy cost of auto-tuning by up to 21.8% and 40.9%, respectively.

Section II presents the background; Section III analyzes DL optimization performance and energy cost; Section IV details the Trimmer auto-tuner design; Section V presents the evaluation results of Trimmer; Section VI discusses related work; and Section VII our research conclusions.

II. Background

A. Cloud Datacenters for Deep Learning

Deep Learning (DL) models: DL Models provide cognitive-like capabilities in areas such as image recognition or language learning [13], [14]. DL models composed of multi-layered Deep Neural Networks (DNNs) are represented as Directed Acyclic Graphs (DAG), where nodes represent DL *operators* (Convolution, Batch Normalization) and edges the operator data dependencies. DL operators are expressed as *tensor programs* comprising both CPU code (e.g. data fetching, submission) and accelerator code (e.g. GPU kernels) that perform tensor manipulation instantiated and executed within a target-device as an *implementation*. DL models with greater number of large operators exhibit improved accuracy by performing a higher number of Floating Point Operations (FLOP) [15].

DL Cloud datacenters: Creation of new DL models, data volume growth, and user demand has resulted in the formation of Cloud datacenters dedicated to provisioning MLaaS as shown in Figure 1. These datacenters are formed by clusters of machines equipped with accelerator devices such as Graphical Processing Units (GPUs) to perform DL model inference (encompassing tensor program execution) substantially faster than conventional CPUs. With production systems such as Facebook performing trillions of DL inferences daily [2], it is paramount that DL-focused Cloud datacenters achieve high performance model inference to satisfy Service Level Agreement (SLA) and maintain Quality of Service (QoS) in a cost-efficient manner. Such cost comprises compute resource (CPU, GPU, network) and time that together drive system energy consumption: a critical issue due to high monetary and environmental cost of datacenter operation [16].

B. DL Model Optimization

Attaining high performance and cost-efficient DL model inference for Cloud datacenters is achieved via DL model optimization to reduce tensor program latency in a target-device [8]. Such optimization is attained by leveraging framework specific tensor program implementations [17], [18] or DL compilers such a TVM [3]. DL compilers enable compilation of high-level DL models definitions onto device-specific binaries providing greater control over implementation behavior. These optimizations consist of both *high-level*: target-device independent transformations of DL model structure (e.g. operator fusion, algebraic simplification, data buffer reuse), and *low-level*: target-device dependent tensor program transformations (alignment to device cache size, tensorization, mapping tensor compute regions [3]). Manually performing tensor program optimization is a time-consuming task that must be performed for hundreds of tensor programs per target-device, and produces sub-optimal tensor program latency improvement [9].

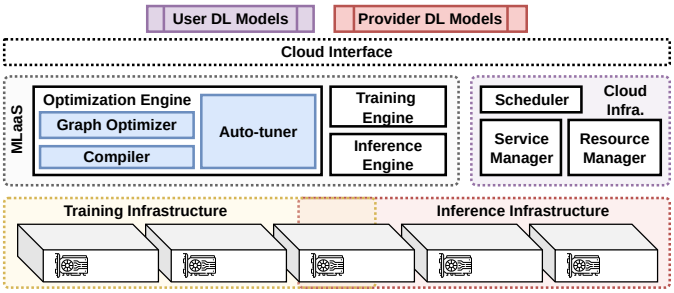


Fig. 1: MLaaS Cloud Infrastructure

C. Auto-Tuning

Auto-tuning is a method for automatic DL model optimization of tensor program implementations towards a target-device. Auto-tuning automates low-level optimization by: (1) searching through a space of possible implementation parameters (i.e. loop tiling extents, unroll factors) known as *template-based* auto-tuning (e.g. AutoTVM, Chameleon [8], [10]); or (2) *autoscheduling*: automatic generation of operator implementations given a set of rules, producing different versions of low-level Intermediate Representation (IR) [4], [9]. Both approaches propose *candidates* - operator implementations in IR compiled towards target-device languages (C, CUDA, x86, PTX), assembled into binaries and executed in isolation to measure their latency. Latency measurements are used to re-train auto-tuner cost models [8], [19] that propose new candidates which are leveraged by search algorithms [3], [10] to navigate the implementation space, avoiding candidates that exhibit high latency.

D. Auto-tuning Cost in Cloud Datacenters

Auto-tuning is a time and energy intensive process, where even small DL models [20] require $\gtrsim$ 10 hours to achieve sizable latency improvements for a single target-device [3], [10]. During auto-tuning, the platform CPU experiences heavy load spikes resulting from candidate space search, code lowering, and compilation of tensor programs. Furthermore, due to the tensor program implementation space is particularly large and non-linear for DL models with

TABLE I: Hardware setup

Abbrev.	Hardware Specification
A	2x (16-core) Intel Xeon 5218 [2.3GHz], 196GB DDR4, Nvidia V100 (Volta) 32GB
B	(6-core) Intel i7-8700K [3.7GHz], 16GB DDR4, Nvidia GTX2080 (Turing): 8GB
C	(12-core) AMD Ryzen 1920X [3.5GHz], 128GB DDR4, Nvidia GTX2080 (Turing): 8GB
D	(6-core) Intel i7-6850K [3.8GHz], 32GB DDR4, Nvidia GTX1080 (Pascal) 8GB

TABLE II: Software and workload setup.

Type	Software Specification	DL Auto-Tuner
Support	Ubuntu 20.04, Docker 20.10.7	RD: Random
Compute	CUDA 11.3.1 [21]	AT: AutoTVM [8]
Driver	Nvidia Driver 465.31	GA: Genetic [8]
Library	Pytorch [18],Apache MXNet [17]	GR: Grid [8]
Compiler	TVM 0.8 & LLVM 11 [3], [22]	

Type	Workload Specification
DL Model	MobileNet-V1/V2 [23], ResNet-18 [20], DenseNet-121 [24], VGG-13/16/19 [13], {batch 1,*3x224x224*}

many complex operators, auto-tuning entails repeatedly executing thousands of candidate tensor programs in an isolated target-device to ensure accurate latency measurement. With the growing number of larger DL models, auto-tuning engages the host platform and target-device for extended periods of time. This is exacerbated by the need to repeat auto-tuning for any architectural changes to the DL model and target-devices.

These aforementioned issues are amplified when considering the scale of Cloud datacenters, whereby DL providers will manually or automatically (via auto-tuning) optimize the many DL models underpinning MLaaS, as well as allow users to directly perform auto-tuning (Amazon SageMaker Neo [5], Alibaba MNN [6], Glow [7]). Thus whilst DL model optimization is effective at reducing the model inference latency for MLaaS, auto-tuning requires a considerable time, compute resources, incurring subsequent energy costs to complete. These optimization costs result in reduced availability and system throughput for DL-focused Cloud datacenters, and incur higher energy consumption - a barrier towards designing sustainable AI infrastructure.

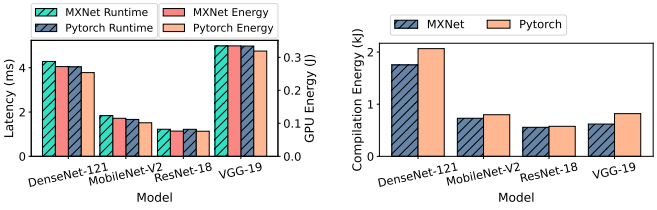
III. DL Optimization Performance & Cost Study

A. Analysis Setup

We have conducted an experimental study pertaining to the performance improvement and energy cost of various DL models when applying different optimization techniques.

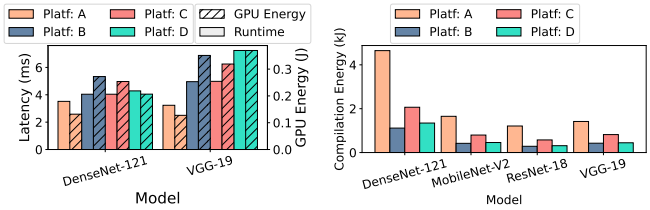
Deployment: Several prominent, well established CNN DL models [25] were examined using a variety of DL frameworks, target-devices and platforms as shown in Tables I & II. We applied both graph-level optimization and auto-tuning, comparing with both baseline implementations and pre-optimized configurations. In all cases, we configured auto-tuners and graph optimizers as reported in related publications and code bases from [26].

Measurement: We measure DL model latency and energy consumption after applying optimization techniques and quantify their cost implications. We observe GPU and CPU power dissipation using a custom profiler based on Nvidia's Managegement Library (NVML) [27] and RAPL/MSR interface for Intel/AMD CPUs. Albeit not explicitly stated in prior studies, all experiments were conducted using graph-level optimization level 3 (TVM), with exception to studying graph optimization levels.



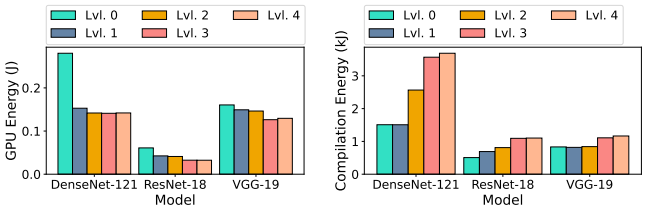
(a) Latency & Run-time Energy (b) Compilation Energy

Fig. 2: DL model latency, run-time & compilation energy with different DL libraries [opt. level: 3]



(a) Latency & GPU energy (b) Compilation Energy

Fig. 3: DL Model latency, run-time & compilation energy with different platforms [opt. level: 3]



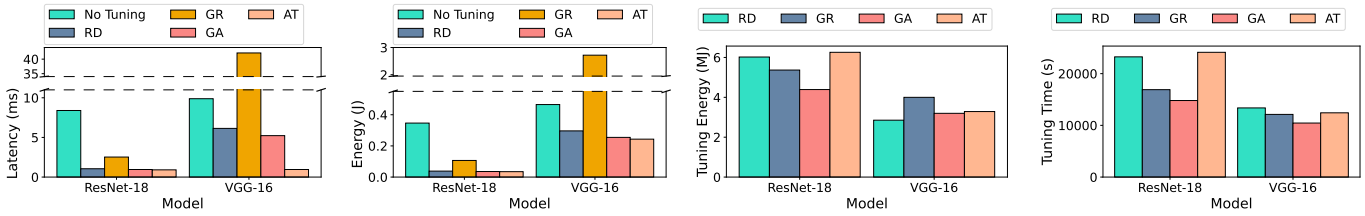
(a) GPU Energy (b) Compilation Energy

Fig. 4: DL Model run-time & compilation energy with varied graph optimization levels. [framework: MxNet / platform: A]

B. Library, Platform, and Graph Optimization

Library: As shown in Figure 2a & 2b, the majority of DL models converted from PyTorch to TVM IR perform on average 1.05× faster than MXNet, however incur a 1.11× mean energy increase during compilation, with a strong positive correlation (Pearson: 0.9987) between model latency and energy. Differences amongst DL frameworks stem from varied DL model IR and compiler parsing approaches; i.e. the compiler maps equivalent operator definitions due to lack of direct matches within the operator library.

Platform: As shown in Figures 3a & 3b, DL model latency and energy profiles vary across platforms and target-devices. Observably, compiling DenseNet-121 on platforms *A* and *C* consumed up to 4.16× more energy compared to *B*, despite slower CPUs with higher number of cores and lower Thermal Design Power. Unique CPU and target device combinations during compilation and model



(a) Latency (ms) (b) Runtime Energy (J) (c) Total Tuning Energy (MJ) (d) Total Tuning Time (s)

Fig. 5: Energy and performance impact of auto-tuning: [Platform: *A*], [HW measurements: 500], [Graph opt. level: 3]

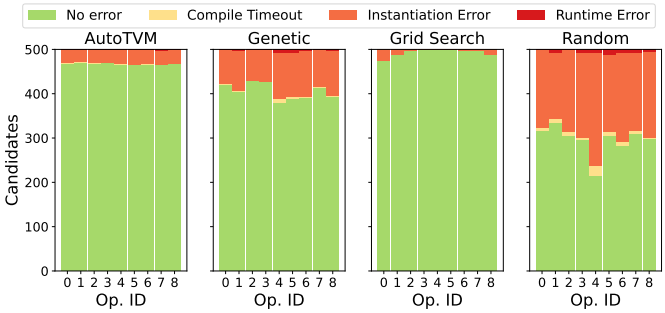


Fig. 6: Error occurrences across different auto-tuners [Hardware measurements: 500, Platform: *A*, Model: ResNet-18]

TABLE III: Graph-level optimization primitives

Lvl.	Primitives
0	Simplify/Partition Graph, Simplify Expressions, Infer Types
1	Fuse Operators, To BBNF, To ANF, To GNF, Eliminate Dead Code, Partially Evaluate, Inline Ops
2	Fold Constants, Split Arguments, Lazy Gradients, DynToStat
3	Canonicalize Operators, Forward/Backward Scale Axis, Eliminate Common Sub-expressions
4	Combine Parallel: Conv2D, Dense, Matmul; Math Approx.

execution exhibit different cost and performance patterns dependant on the platform composition, the choice of DL framework, model architecture, operator computational complexity, as well as the suitability of tensor program implementations towards given target-device.

Graph: As shown in Figure 4a, applying further graph optimization levels (see. Table III) decreases model latency and energy consumption by 25-50%, yet also increases compilation energy by 29-60% for levels 0 to 4 as shown in Figure 4b. This stems from strong correlation between model latency, energy cost, and computation required to apply consecutive graph optimizations. We observe that latency improvements at the same optimization level vary across models; i.e. applying level 4 to VGG-19 improved latency by 22.5% while for DenseNet-121 a 50.1% improvement, however required 60.6% more energy during compilation.

C. Auto-tuning

Performance: Our experiments indicate that the choice of auto-tuner alters latency improvement and energy profiles of DL models, as shown in Figures 5a & 5b. Compared to a baseline model (model compiled with default operator implementations), sophisticated auto-tuners (AutoTVM, Genetic Search) result in latency decrease of 9.22 - 10.35× and energy cost reduction of 1.91 - 9.98× (Figure 5b). In contrast, brute-force auto-tuners (Random, Grid Search) can result in moderately faster or sometimes slower latency compared to baseline as per Figure 5a. Sophisticated auto-tuners explore implementation spaces more efficiently, using cost models to guide search towards faster candidates.

TABLE IV: Cold candidate impact across platforms, average of all tuned operators when tuned with Auto-TVM

Model	Platform	Num. Cold	Time Cold %	Energy Cold
ResNet-18	A	133±21	26.74±4.3	52.1±10.4kJ
	B	134±18	26.86±2.8	36.7±7.9kJ
	C	130±11	26.13±2.2	37.1±4.2kJ
	D	139±14	27.92±2.9	29.0±4.6kJ
VGG-16	A	202±29	40.51±5.9	76.7±2.0kJ
	B	201±28	40.24±5.6	49.3±2.4kJ
	C	200±29	40.16±5.9	49.1±4.2kJ
	D	216±25	43.33±5.0	41.2±3.1kJ

Energy: Similar to DL model latency, there exists a strong positive correlation between auto-tuning time and energy cost (0.9880 Pearson coefficient), albeit different auto-tuners incur varied time and energy costs across DL models as shown in Figures 5c & 5d. When applied to ResNet-18, AutoTVM produced the lowest latency of 0.89ms, however incurred an additional 7200–8300s and 240–1800kJ compared to Grid Search and Genetic Search, which stems from querying and updating AutoTVM's cost model and performing Simulated Annealing. Surprisingly, Random Search was significantly costlier (23,250s & 6,020kJ) compared to other approaches due to the random implementation space traversal, inadvertently compiling and executing both slow and erroneous candidates significantly increasing cost. The observed differences in cost amongst auto-tuners stem from operator heterogeneity, implementation space size, and auto-tuners varying effectiveness at space traversal and sampling promising candidates.

Failed Candidates: We discovered that all auto-tuners experienced a sizeable number of compilation and run-time errors when generating candidates as shown in Figure 6. Instantiation (compilation) errors were the most common failures observed, caused by stochasticity of auto-tuners, particularly in Genetic (15%), AutoTVM (7.2%), and Random (36%). Run-time errors and timeouts represented further 1.9% and 2.6% of failures. Existing auto-tuners determine when to stop optimizing based on the total number of performed measurements (inclusive of failures) requested by the user. Whilst erroneous candidates only consumed

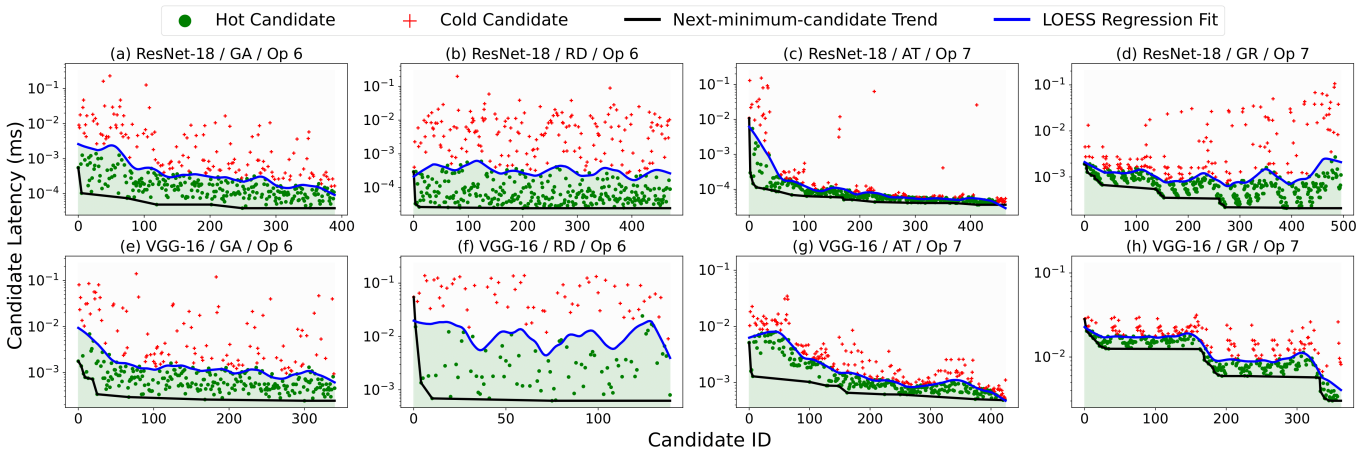


Fig. 7: Performance trends of different tuning frameworks with 500 HW measurements, depicting: best-case performance trend, LOESS regression of candidate performance, *hot* candidates and *cold* candidates

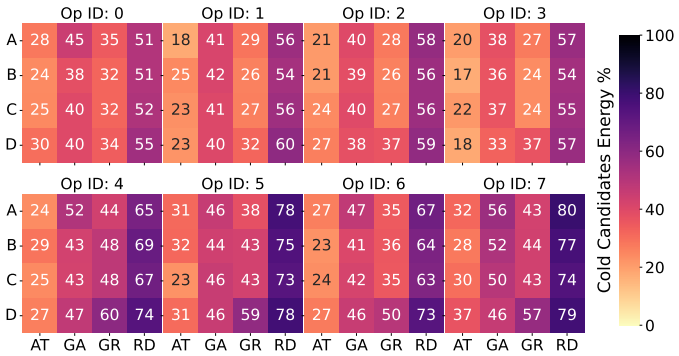


Fig. 8: Percentage of energy consumed by *cold* candidates across eight Conv2D operators of ResNet-18.

additional 11-18% energy, they reduced the opportunity for desirable candidate exploration and incurred additional costs; thus reducing auto-tuning effectiveness and cost-efficiency.

Cold Candidates: A sizeable portion of candidates exhibit high latency, yet do not meaningfully contribute to optimization progress. We categorize these so called *cold* candidates by applying LOESS regression [28]. With higher latency, cold candidates incur more auto-tuning cost, as shown in Figure 7 and Table IV. We observe that AutoTVM produced the most cold candidates early on due to cost model initialization, where Grid and Random Search strategies exhibit larger diffusion. Particularly problematic given their time and energy costs, cold candidates contributed on average to 50.5% of total auto-tuning cost, as shown in Table IV, varying across operators and platforms, as shown for ResNet-18 in Figure 8. Cold candidates produced by AutoTVM were responsible for 17-38% of total auto-tuning energy cost, and 80% in worst case for Random Search. As per Table IV, the choice of platform had impact proportional to its compute capabilities.

Convergence: The choice of an auto-tuner determines optimization convergence patterns, indicated by the *next minimum candidate* trend line shown in Figure 7. More sophisticated frameworks (AutoTVM, Genetic Search) converge early compared to Grid Search or Random Search. Crucially, whilst most frameworks progressively propose faster candidates, we observed that *hot* candidates (low latency candidates that contribute positively to auto-tuning progress) could be found relatively early during auto-tuning. For example, as shown in Figure 7, auto-tuning Operator 7 with AutoTVM discovered candidates merely 100ns (3-5%) slower than globally fastest candidate within the first 87 out of 500 total measured candidates.

D. Design Directions

From our study, we identify multiple important design descisions required for cost effective DL auto-tuning.

(1) Understand energy diversity: We observe that the interplay between DL models, frameworks, auto-tuners, target-devices and platforms uniquely impacts the cost and performance of auto-tuners, while existing works focus primarily on reducing cost w.r.t. target-devices [3], [10]. We also observe that no single optimization approach exhibits a guaranteed latency improvement at reduced cost. Leveraging these insights is useful to *determine models and optimizers that work well for specific target-devices and host platforms.*

(2) Avoid erroneous and cold candidates: Observably, all examined auto-tuners generate erroneous candidates, which do not contribute towards performance convergence nor explore favourable candidates. We also observe that *cold* candidates exhibit high operator latency without contributing towards successful candidate selection, incurring high time and energy cost. Avoiding both erroneous and cold candidates is useful to *reduce optimization costs whilst maintaining reasonable latency improvements.*

(3) Leverage hot candidates: Our analysis suggests that candidates with acceptably low latencies can be found early during the candidate search. Whilst increased number of measurements results in operator latency improvement, it is possible to ascertain *hot* candidates (exhibit low latency and positively contribute to convergence) soon after auto-tuning commences. Leveraging hot candidates could help to *rapidly determine the efficacy of an optimization technique.*

IV. Trimmer Framework

A. Overview

Trimmer is designed to improve the cost-efficiency of auto-tuning and operates as a component for optimizing trained DL models that are ready to be deployed within a Cloud datacenter, as shown in Figure 9. Such models originate from both the provider (as part of their MLaaS offering) and users (submitting models for training or deployment). Trimmer achieves fast and low-cost auto-tuning via enhancements at both operator and DL model levels.

At operator-level, Trimmer performs neural network based *cold* candidate filtering that predictively excludes candidate implementations that exhibit poor performance (*cold* candidates) and re-sampling for more favourable candidates (*hot* candidates) to accelerate auto-tuning convergence onto sufficiently optimal implementations quicker. At DL model-level, Trimmer performs *Survey-tuning* whereby each operator is partially optimized using a small number of hardware measurements and periodic latency measurement of the complete DL model, thus allowing early completion based on measured latency improvements. Combining these approaches, Trimmer can optimize models simultaneously across a Cloud cluster with multiple hardware platforms, ranking their progress and suspending optimization based on comparative latency improvements. We implement this functionality using Docker and leverage custom RPC-based routines to execute model auto-tuning remotely.

Formally, Trimmer's goal is to minimize DL model inference latency f_t by auto-tuning each operator fo_i 2 $Oji = 1$ Kg, where K is the number of operators in the model. The amount of time spent on optimizing each o_i depends on its rate of its latency improvement relative to other o_j in the model and overall improvement to model latency. Overall, Trimmer's objective is to optimize the DL model f_t , given a (user-specified) latency goal g, where g latency of the unoptimized model.

$$\min \sum_{i=1}^{K} o_i \quad \text{s.t. } f_o \quad g \tag{1}$$

B. Cold Candidate Filtering

Inspired by learned index structure approaches [29], we created a DL model and leveraged outputs of its intermediate layer to query candidates based on their similarity, and filter out top-n *cold* candidates ahead of hardware measurement.

Overview & Training: We designed a three-layer fully connected (FC) network with ReLU activation function [30] as outlined in Table V. We implemented and trained our model using PyTorch [18]. During training of our FC network, we minimize the Mean Squared Error (MSE) against individual operator latency, using 20,000 data points collected during our study (Section III-C) as the dataset. We prepare our dataset by splitting it into <train:validation:test> sets with a 7:1:2 ratio and leverage the validation set for hyperparameter tuning following prior work in predicting

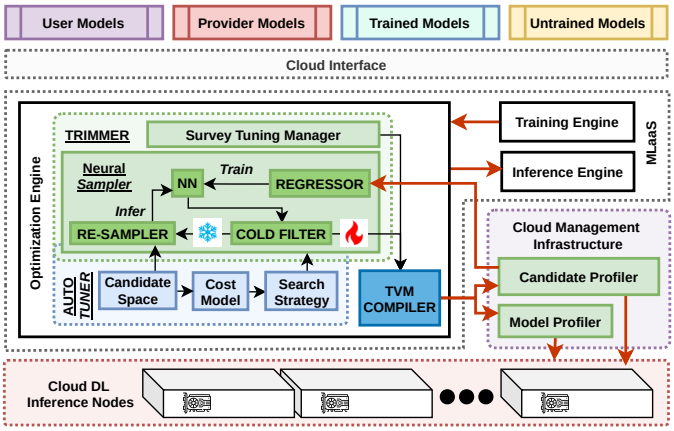


Fig. 9: Trimmer system architecture

TABLE V: FC Network Architecture. Tuple (x,y) is layer dimension, x and y are the number of input and output neurons, respectively.

Layer Type	Layer Dimensions
Embedding e_i	(R_{space}, 10)
Input	($\sum_{i=1}^{E} e_i$, 32)
Middle	(32, 32)
Output	(32, 1)

operator latency [31]. Our model was trained for 10 epochs utilizing the Adam Optimizer [32], with batch size 1024, learning rate of $1e^{-3}$ (min $1e^{-9}$) and Plateau Patience of 1. The model learns relationships between operator latency and the operator, candidate and target-device characteristics.

Model Inputs: To capture non-linearity of the mentioned characteristics, the input to our FC network is a vector that includes: (1) implementation candidate configuration, where each entry is an integer describing low-level optimization parameters of the configuration; (2) features of the target-device and its host platform; (3) the theoretically achievable operator FLOPs; (4) a unique operator identifier; and (5) a unique representation of operator arguments (for convolution - kernel size, padding, stride...). During filtering we extract the number of outputs R_{space} for each operator configuration by accessing its total *configuration space* and construct an *embedding*, where each integer entry is a vector of parameters of size k, and k is a hyperparameter that we set to 10 based on empirical findings. Categorical features such as target-device or operator characteristics are transformed into integers and have their own embeddings. We then feed the concatenated features into our FC network.

Exploration: As described in Section III-C, it is beneficial to reduce costs associated with measuring *cold* candidates, however, the auto-tuner should explore a wide range of points in the implementation space to identify *hot* candidates. To achieve this, Trimmer uses an inverse ϵ-greedy strategy to incentivize exploitation of filtered candidates early on during auto-tuning and progressively reduces the probability of candidate filtering after each cost model update, such that space exploration is favoured as soon as auto-tuner search strategy and cost model are stable.

Algorithm 1 Cold Candidate Filtering

Input: ($Samples$, ϵ, k, $model$)

// batch inference the task configuration samples
$embeddings$ MODEL.PREDICT($Samples$)
// for each candidate task's embedding
for e in $embeddings$ **do**
  $rand$ RAND
  // ϵ is a decreasing parameter
  if $rand < \epsilon$ **then**
    // return the number of cold candidates within k similar samples
    o SIMILARINDATABASE(k, e)
    if $o == k$ **then**
      // remove sample from the Set S
      REMOVE(s)

Querying configuration: Trimmer heuristically prunes the proposed candidates if the top-n similar candidates are *cold* candidates, as described in Section III-C. We leverage the middle layer output of our FC network to predict candidate latency by querying top-n similar configurations within an operator tuning database using Cosine Similarity (commonly used to identify similar samples in a vector space [33]). We replace the pruned high-latency candidates with new unseen samples to ensure effective exploration. These probabilistic and heuristic strategies are then combined in a single procedure, as shown in Algorithm 1.

C. Survey Tuning

Existing auto-tuners optimize operators sequentially, performing N hardware measurements before optimizing the next operator. Auto-tuning is considered complete once all operators have been optimized fully (up to specified number of hardware measurements N), as shown in Figure 10. With this assumption, latency of a DL model can be ascertained only after all operators are optimized, with the user having no knowledge of achieved DL model latency whilst auto-tuning is underway. Existing auto-tuners enable early stopping of optimization by an arbitrarily set threshold, however, it is performed on a per-operator basis, with no means to compare improvements relative to other operators. Our study in Section III-C suggests that well-performing *(hot)* candidates can be found relatively early during the auto-tuning process. We leverage this in Trimmer and reduce optimization cost by providing *Survey-tuning*, as depicted in Figure 10, both at operator and model level.

Operator Level. Trimmer places all tunable operators into a queue and performs auto-tuning in *batches* of n hardware measurements. After a complete tuning *epoch* (i.e. each operator has completed a batch of measurements), we compile and evaluate the DL model using best candidate operator configurations found so far. This enables Trimmer to suspend or early-stop auto-tuning based on the trade-off between projected model latency improvement and optimization cost so far. The early-stop or suspension mechanism uses achieved model latency as goal g. If goal g is not reached, or not reached within a desired cost budget (time, energy, monetary...), Trimmer compares the rate of latency reduction compared to last batch, as:

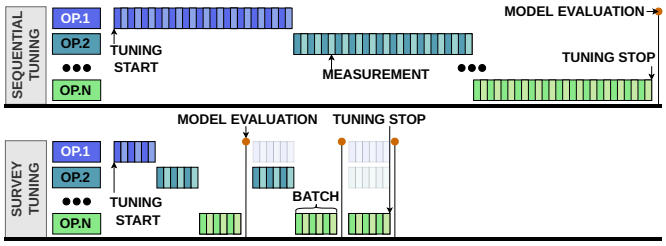


Fig. 10: Sequential and Survey tuning, [OP = Operator].

$$stop = \omega_1 \frac{x_t + x_{t-1}}{x_{t-1}} \leq \phi \tag{2}$$

$$x_t = \omega_2 \frac{\delta f_t}{\delta O_t} \tag{3}$$

Where x_t is the ratio of difference in model latency f between t and t 1 over the auto-tuning time O_t for the epoch. To account for the cases (1) where both x_t and x_t 1 are positive (slower), and (2) where x_t is faster but x_t 1 was slower, ω_1 is set to 1 and 1 otherwise; ω_2 is set to 1 if both δf_t and δO_t are negative, 1 otherwise. These cases occur due to high non-linearity of the operator candidate space that may cause the auto-tuning to diverge in discovered latency abruptly. The intuition is that Trimmer can maximize optimization efficiency by checking whether the model latency is decreasing every epoch at a sufficient rate given its time and energy cost, where ϕ is a hyperparameter that specifies whether the change in the ratio should be greater than ϕ% to avoid excess tuning, where the incremental performance change between models per batch is not significant. This hyperparameter allows practitioners to prioritize either achieved performance or cost accordingly. In our case, we have set ϕ to -0.25 providing an appropriate balance between performance improvement and cost in optimization, motivated by empirical study.

Model Level: The suspension approach described above can also be applied at model level (measuring model latency at each batch). This transition from per-operator to per-model allows for meta-tuning, where the relative speed-up of models is compared to concurrent auto-tuning processes of other models within a cluster of machines. Inspired by [34], we implement multi-model auto-tuning by ranking models by their most recently achieved latency improvement in an auto-tuning batch, and suspend models that perform the worst. The intuition behind this population-based approach is to focus optimization effort onto the most promising models when tuning multiple models within the cluster; and depending on user set criterion of latency threshold or cost objective. The suspension interval can include a number of configurable plateau iterations (i.e. poor performance improvement in consecutive batch intervals). We synchronize multi-model auto-tuning processes across a cluster of machines at intervals of completed batches of individual models - permitting each model to be optimized for several batches before being compared.

TABLE VI: Single Platform evaluation results (latency, auto-tuning time and total energy consumed during auto-tuning

	Model	TR	AT	RL	CH	Base.
Latency (ms)	Alexnet	**0.79** 0.02	**0.85** 0.05	**0.84** 0.09	**0.82** 0.08	4.42
	VGG-16	**4.68** 0.49	**4.78** 0.28	**5.85** 0.45	**5.83** 0.38	9.66
	Mobilenet	**0.65** 0.03	**0.67** 0.02	**0.76** 0.06	**0.74** 0.05	1.24
	ResNet-18	**1.39** 0.29	**0.86** 0.08	**1.11** 0.06	**1.03** 0.08	8.48
Tuning Time (m)	Alexnet	**119** 0.29	**116** 0.25	**121** 0.42	**127** 0.40	
	VGG-16	**194** 0.49	**207** 0.22	**296** 0.98	**298** 0.30	
	Mobilenet	**213** 0.48	**286** 0.52	**216** 0.58	**214** 0.42	
	ResNet-18	**228** 0.24	**401** 0.59	**353** 0.74	**279** 0.44	
Tuning Energy (MJ)	Alexnet	**1.6** 0.22	**1.9** 0.22	**2.3** 0.46	**2.4** 0.50	
	VGG-16	**3.4** 0.59	**3.4** 0.21	**5.5** 1.19	**5.6** 0.32	
	Mobilenet	**3.6** 0.81	**4.5** 0.48	**3.6** 0.58	**3.6** 0.41	
	ResNet-18	**21.2** 1.6	**26.6** 2.4	**29.5** 3.4	**31.8** 3.1	

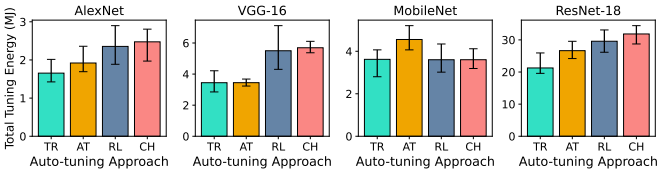


Fig. 11: Total auto-tuning energy consumption

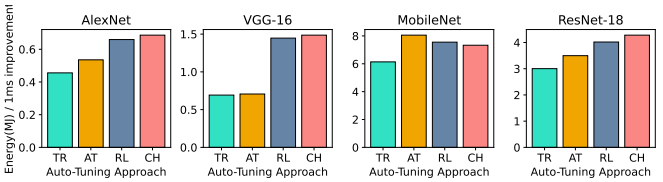


Fig. 12: Ratio of average total energy spent (MJ) during auto-tuning to achieve 1ms of model latency improvement

V. Evaluation

A. Setup

Environment: We utilized four unique DL models and performed our experiments on hardware platforms described in Table I. Where appropriate, models selected for optimization share identical model configurations and hyperparameters with prior work [8], [10], and were converted from Pytorch using TVM. Throughout our evaluation, we selected graph optimization level 3 in line with code bases online to avoid value approximation optimizations at higher levels (see Table III) that may affect model accuracy. In this work, we also assumed access to an offline database of historical optimization data. In the case where there is no historical data, auto-tuning can be executed to collect n iterations of data for our FC network training.

Auto-tuners: We compared Trimmer (TR) against three state-of-the-art auto-tuners: *AutoTVM* (AT): utilizes XGBoost to avoid excessive hardware measurements and Simulated Annealing (SA) optimizer to search through the candidate space given feedback from the cost model [3]; *Reinforcement Learning* (RL): uses the XGBoost cost model and Proximal Policy Optimization [10] and Q-Learning [11] as the optimizer to propose candidates; and *Chameleon* (CH): extending the RL approach, and uses K-means clustering to reduce similar candidates in the search space to accelerate auto-tuning [10]. The auto-tuners were configured to perform 500 hardware measurements (candidate batch size = 64), as per default configurations of the respective auto-tuners.

Metrics: We measured effectiveness of Trimmer using DL model latency (before and after applying auto-tuning), as well as total optimization time and energy consumption that constitute the auto-tuning cost per model. We also monitored the platforms under test for CPU and GPU metrics such as utilization or memory usage and measured any resource overheads incurred from Trimmer's mechanisms.

Experiment scenarios: We have conducted experiments that evaluate Trimmer w.r.t. reduction in, and cost-efficiency of, auto-tuning time and energy cost. We evaluated Trimmer on *Single-platform* DL model auto-tuning, performing isolated experiments on individual target-devices, as well as *Cloud cluster* where we coordinate four auto-tuning instances across four platform *A* machines to compare Survey tuning against sequential auto-tuning in parallel.

B. Experiment Results: Single Platform, Sequential Tuning

Performance: Trimmer achieved a greater model latency reduction compared to other auto-tuning approaches for the majority of scenarios. As shown in Table VI, Trimmer achieved the fastest inference time at 0.79ms, 4.68ms and 0.65ms for AlexNet, VGG-16, MobileNet, respectively. In the case of ResNet-18, Trimmer achieved comparable results to RL and Chameleon approaches. The reason for these results is due to Trimmer's cold candidate filtering, allowing for quicker identification of *hot* candidates relatively early within the auto-tuning process. Such phenomena is observable when inspecting the shape of operator candidates measurement patterns as shown in Figure 13.

Cost-efficiency: As shown in Table VI, Trimmer completed auto-tuning with a lower time cost than other auto-tuners for VGG-16, ResNet-18, and performed within margin of error of the fastest framework for remaining models. Moreover, Trimmer's system energy consumption is lower during auto-tuning for AlexNet and ResNet-18 and scored on par with the least energy-hungry framework (AutoTVM) in remaining models cases. Crucially, such auto-tuning costs should be considered in the context of their cost-efficiency (i.e. when an auto-tuner produces low model latency, yet consumes significantly more energy). Across all models, Trimmer achieves the highest cost-efficiency for the *amount of energy required to achieve 1ms latency reduction* as depicted in Figure 12 with an average improvement over other auto-tuners of 14-33% (AlexNet), 2-54% (VGG-16), 16-24% (MobileNet) and 14-29% (ResNet-18). This stems from cold candidate filtering and early suspension upon detection of insufficient latency reduction, and is linked to the design of our neural sampler, which was trained on a range of model and target-device samples as per Table I.

Candidates: Our results suggest that the auto-tuner cost model guided by Trimmer proposes fewer globally poor and failed candidates compared to RL and Chameleon, and on average explores more candidates than AutoTVM within the

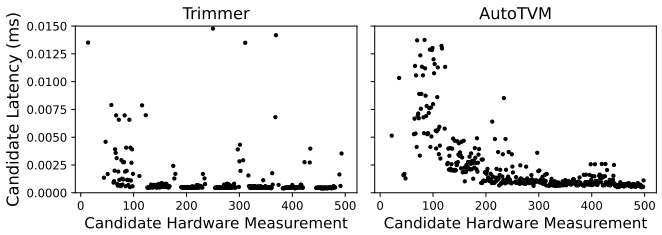
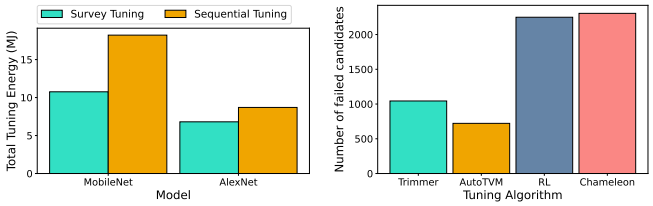


Fig. 13: Comparison of candidate hardware measurement patterns in Trimmer and AutoTVM in VGG-16 (Operator-5)



(a) Total tuning energy (b) Total failed candidates

Fig. 14: Total tuning energy & number of failed candidates when tuning VGG-16 *intel.v100* – Survey vs. sequential in parallel

same time (see Figure 14). This stems from the design of the Trimmer re-sampling mechanism, which filters out cold candidates from the currently measured batch, whilst greedily reintroducing a portion of new candidates proposed by the cost model to diversify the batch. This produces more cost-efficient exploration (as more overall candidates are explored) and inadvertently explores more cold candidates compared to filter-less AutoTVM approach. The additional processing of candidate filtering resulted in 3% CPU utilization increase compared to AutoTVM, which given the latency improvement and cost-efficiency attained by Trimmer, we deem an acceptable resource cost overhead.

C. Experiment Results: Cloud Clusters

Performance & Energy: Trimmer was able to achieve a 21.8% and 40.9% reduction to total auto-tuning energy cost from Survey tuning in comparison to parallel auto-tuning for AlexNet(1.8MJ) and MobileNet(5.97MJ), respectively as shown in Figure 14. The reason for this energy reduction is due to Survey tuning capturing improvements to model inference latency across all operators as well as other models at regular batch interval (and if insufficient performance gains were detected, optimization is suspended). A key advantage of Trimmer Survey tuning is that it allows for strongly performing auto-tuning frameworks to conduct additional batches over less effective algorithms, improving it's cost-efficiency. This is particularly the case for MobileNet where Trimmer achieved an inference time 0.3ms (8%) faster than AutoTVM running to full completion per Table VII.

As shown in Figure 15, when using the Survey tuning to periodically evaluate the model at each batch interval, we observed that each framework exhibited different convergence patterns across both models. For the case of Mobilenet, we observed that whilst all frameworks produced

TABLE VII: Cloud cluster Survey tuning vs. parallel auto-tuning

	Model	Survey	Parallel	Improve
Tuning Energy (MJ)	MobileNet	10.76	18.21	40.9%
	AlexNet	6.80	8.70	21.8%
Model Run-time (ms)	MobileNet	0.629	0.684	8%
	AlexNet	0.797	0.805	1%

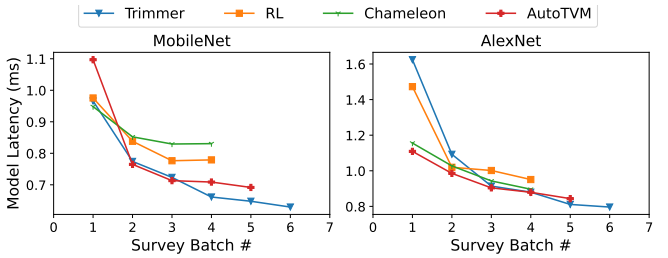


Fig. 15: Survey tuning at different batches (64 candidates per batch)

a 10.1–30.2% inference latency improvement between Batch-0 and Batch-1, both RL and Chameleon were suspended at a relatively early batch interval due to reduced performance improvements detected at Batch-3 – both in terms of its own convergence gradient and in comparison to other frameworks. AutoTVM was suspended at Batch-5 for achieving relatively little improvement over two consecutive intervals, as well as in relation to Trimmer, which was able to produce best inference time at Batch-5 (0.62ms).

These results affirm that Trimmer: (1) enables the auto-tuner search strategy to explore further into the candidate space given the same amount of time and energy, providing more cost-efficient auto-tuning; and (2) via the Survey tuning meta-strategy, Trimmer achieves overall faster optimization times compared to Sequential tuning at operator-level, and Parallel optimization of multiple models simultaneously due to its ability to suspend tuning given less favourable performance improvement across a Cloud cluster. Additionally, given Survey tuning is a meta-tuning mechanism that works in conjunction with the auto-tuner and does not modify its operation and can integrate with any auto-tuner that reports latency measurements in real-time including ones that do not rely on templates for implementation scheduling [9], [11].

VI. Related Work

Cost-efficient DL for Cloud datacenters. Creating cost-efficient DL systems has been gaining traction within the ML community, most notably for system energy consumption [12], [35]. Frameworks designed to improve DL-focused Cloud datacenter cost-efficiency have recently been proposed to accelerate DL model hyperparameter tuning, efficient architecture search and training [36]. Amazon SageMaker [37] focuses on training, hyperparameter tuning and optimization (Neo). Lorien [38] supports tuning of DL models across clusters of machines and collects best performing schedules to achieve faster training of predicting model accuracy.

DL Tensor Program Optimization. Leveraging search algorithms and ML to automatically optimize programs is an

active research area. Search-based approaches have been shown to optimize complex FFT or BLAS routines [39] and perform I/O parameter search [40]. DL compilers increasingly adopt auto-tuning; AutoTVM [8] uses Simulated Annealing and Gradient-boosting to parameterize templates for tensor program implementations. Deep Reinforcement Learning (DRL) is used for implementation search and filtering by both Chameleon [10] and AdaTune [19], whilst Autophase [41] leverages DRL to generate a compilation optimization order. Autoschedulers such as Ansor [9] generate implementations hierarchically and fine-tune their parameters via evolutionary search, whilst FlexTensor [11] explores program space using DRL and heuristics to generate candidates. One-shot-tuner [42] modifies AutoTVM with a Transformer cost model to predict operator performance. Trimmer proposes a sampling mechanism based on an FC neural network to encourage measurements of fast candidates whilst filtering out candidates that are likely to be slow or erroneous. Trimmer also introduces an ϵ-*greedy* meta-tuning strategy – Survey tuning at both operator and model level that reduces auto-tuning cost by enabling early finish, periodically measuring DL-model-level performance. Survey tuning further enables auto-tuning across a Cloud cluster and multiple models simultaneously.

VII. Conclusions

In this paper we propose Trimmer: a DL model auto-tuning framework that performs high-performance and cost-efficient tensor program optimization for Cloud datacenters. We have empirically analyzed the diverse energy and performance characteristics of different DL model auto-tuning and optimization techniques across various hardware devices. Through conducting experimentation we have demonstrated that Trimmer is capable of improving DL model performance whilst reducing auto-tuning energy cost considerably. We hope that our framework provides new insights to aid the study – and creation of – cost-efficient Cloud datacenters designed to provision MLaaS.

References

[1] I. Goodfellow, B. Yoshua, and C. Aaron, *Deep learning*. MIT press Cambridge, MA, USA, 2016.

[2] K. Hazelwood *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *HPCA*, 2018.

[3] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *USENIX OSDI*, 2018, pp. 578–594.

[4] A. Adams *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics*, pp. 1–12, 2019.

[5] A. Inc. (2020) Amazon sagemaker neo. [Online]. Available: https://aws.amazon.com/sagemaker/

[6] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai *et al.*, "MNN: A universal and efficient inference engine," *MLSys*, 2020.

[7] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[8] T. Chen *et al.*, "Learning to optimize tensor programs," in *NeurIPS*, 2018, pp. 3389–3400.

[9] L. Zheng *et al.*, "Ansor: Generating high-performance tensor programs for deep learning," *OSDI*, 2020.

[10] B. H. Ahn *et al.*, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," in *ICLR*, 2020.

[11] S. Zheng *et al.*, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *ACM ASPLOS*, 2020, p. 859–873.

[12] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green ai," *arXiv preprint arXiv:1907.10597*, 2019.

[13] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ICLR*, 2015.

[14] C. Raffel *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *arXiv e-prints*, 2019.

[15] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *PMLR*, 2019.

[16] M. Zakarya, "Energy, performance and cost efficient datacenters: A survey," *Renewable and Sustainable Energy Reviews*, vol. 94, 2018.

[17] Apache Software Foundation. (2020) Apache MXNet | a flexible and efficient library for deep learning. [Online]. Available: https://mxnet.apache.org

[18] PyTorch. (2020) Pytorch. Facebook's AI Research lab (FAIR). [Online]. Available: https://pytorch.org/

[19] M. Li *et al.*, "Adatune: Adaptive tensor program compilation made efficient," *Advances in Neural Information Processing Systems*, 2020.

[20] K. He *et al.*, "Deep residual learning for image recognition," in *IEEE conference on Computer Vision and Pattern Recognition*, 2016.

[21] NVIDIA. (2020) Nvidia cuda. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[22] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008.

[23] A. G. Howard, M. Zhu *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[24] G. Huang, Z. Liu, L. Van Der Maaten, K. Q. Weinberger *et al.*, "Densely connected convolutional networks," in *CVPR*, 2017.

[25] M. Coşkun *et al.*, "An overview of popular deep learning methods," *European Journal of Technique (EJT)*, vol. 7, no. 2, pp. 165–176, 2017.

[26] "github - apache/tvm: open deep learning compiler stack for cpu, gpu and specialized accelerators," 2022. [Online]. Available: https://github.com/apache/tvm/

[27] NVIDIA. (2020) Nvidia management library. [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml

[28] W. S. Cleveland and S. J. Devlin, "Locally weighted regression: an approach to regression analysis by local fitting," *Journal of the American statistical association*, vol. 83, no. 403, pp. 596–610, 1988.

[29] T. Kraska *et al.*, "The case for learned index structures," in *International conference on management of data*, 2018, pp. 489–504.

[30] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[31] L. Zhang *et al.*, "Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices," in *MobiSys*, 2021.

[32] K. Xu *et al.*, "Show, attend and tell: Neural image caption generation with visual attention," in *ICML*, 2015.

[33] N. Dehak, P. J. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet, "Front-end factor analysis for speaker verification," *IEEE/ACM TASLP*, 2011.

[34] M. Jaderberg, V. Dalibard, S. Osindero, W. M. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, "Population based training of neural networks," *arXiv preprint arXiv:1711.09846*, 2017.

[35] T.-J. Yang *et al.*, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *ECCV*, 2018.

[36] D. Golovin, B. Solnik, S. Moitra *et al.*, "Google vizier: A service for black-box optimization," in *ACM SIGKDD*, 2017.

[37] E. Liberty, Z. Karnin, B. Xiang, L. Rouesnel *et al.*, "Elastic machine learning algorithms in amazon sagemaker," in *SIGMOD*, 2020.

[38] C. H. Yu *et al.*, "Lorien: Efficient deep learning workloads delivery," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021.

[39] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[40] J. Ansel, S. Kamil, K. Veeramachaneni *et al.*, "Opentuner: An extensible framework for program autotuning," in *PACT*, 2014.

[41] Q. Huang *et al.*, "Autophase: Compiler phase-ordering for hls with deep reinforcement learning," in *FCCM*, 2019.

[42] J. Ryu, E. Park, and H. Sung, "One-shot tuner for deep learning compilers," in *ACM SIGPLAN CC*, 2022.