Transfer Learning Across Heterogeneous Features For Efficient Tensor Program Generation

Gaurav Verma
Stony Brook University
Stony Brook, New York, USA
gaurav.verma@stonybrook.edu

Siddhisanket Raskar
Argonne National Laboratory
Lemont, Illinois, USA
sraskari@anl.gov

Zhen Xie
Argonne National Laboratory
Lemont, Illinois, USA
zhen.xie@anl.gov

Abid M. Malik
Brookhaven National Laboratory
Upton, New York, USA
amalik@bnl.gov

Murali Emani
Argonne National Laboratory
Lemont, Illinois, USA
memani@anl.gov

Barbara Chapman
Stony Brook University
Stony Brook, New York, USA
barbara.chapman@stonybrook.edu

ABSTRACT

Tuning tensor program generation involves searching for various possible program transformation combinations for a given program on target hardware to optimize the tensor program execution. It is already a complex process because of the massive search space, and exponential combinations of transformations make auto-tuning tensor program generation more challenging, especially when we have a heterogeneous target. In this research, we attempt to address these problems by learning the joint neural network and hardware features and transferring them to the new target hardware. We extensively study the existing state-of-the-art dataset, TenSet, perform comparative analysis on the test split strategies and propose methodologies to prune the dataset. We adopt an attention-inspired approach for tuning the tensor programs enabling them to embed neural network and hardware-specific features. Our approach could prune the dataset up to 45% of the baseline without compromising the Pairwise Comparison Accuracy (PCA). Further, the proposed methodology can achieve on-par or improved mean inference time with 25%-40% of the baseline tuning time across different networks and target hardware.

CCS CONCEPTS

• **Software and its engineering → Compilers**; • **Computing methodologies → Machine Learning; Artificial intelligence**.

KEYWORDS

auto-tuning, deep learning compilers, heterogeneous transfer learning, tensor program generation

ACM Reference Format:
Gaurav Verma, Siddhisanket Raskar, Zhen Xie, Abid M. Malik, Murali Emani, and Barbara Chapman. 2023. Transfer Learning Across Heterogeneous Features For Efficient Tensor Program Generation. In *Venue, TBD*. ACM, New York, NY, USA, 7 pages. https://doi.org/xx.xxxx/x.y

ExHet'23, February 25th, 2023, Montreal, Canada

ACM ISBN xxx-x-xxxx-xxxx-dd/yy/mm…$15.00
https://doi.org/xx.xxxx/x.y

1 INTRODUCTION

Deep neural networks (DNN) applications are ubiquitous across multiple artificial intelligence domains, including industrial and scientific disciplines. They form the backbone of many existing and emerging applications. The DNN development is rapidly advancing due to the the capabilities of computing hardware and domain-specific accelerators that make the execution of tensor programs efficient by providing hand-tuned deep learning (DL) libraries. These libraries are often not scalable. Tensor compilers such as XLA [13], TVM [3], and Glow [11] facilitate fine-grained hardware-independent (high-level) and dependent (low-level) optimizations to the input computation graphs. A tensor compiler analyzes the computation graphs and applies various optimizations at different stages. Additionally, it inputs subgraphs or mathematical expressions and selects the optimized low-level implementation generating a tensor program. Tensor program generation refers to automatically generating code for tensor programs.

The deep neural architecture has evolved manifolds from simple neural networks to convoluted ones, followed by recurrent ones to massively large models.Advancements in hardware such as GPUs and domain-specific accelerators and DL frameworks like TensorFlow and PyTorch offer optimized kernel support facilitating DL innovations. With advances in DNN architectures and backend hardware, the search space of compiler optimizations has grown manifold. The vast search space consisting of loop optimizations like tiling, vectorization, etc., limits the use of data-driven approaches to auto-tune the tensor compilers and generate efficient tensor programs. With automatic tuning of tensor compilers to generate performant kernel plans becoming prevalent, proposed methodologies based on data-driven cost modeling and intelligent techniques require significant amounts of hardware data to learn cost models. Often, these datasets are specific to the nature of the experiments in the HPC domain. These resource-intensive techniques are hindered by the large number of kernels required upon introducing new hardware. In such a scenario, the cost model or a tuner requires retraining from scratch.

Additionally, almost all of these cost models [4, 12] are based on the train and test data drawn from identical probability distributions, while the source and target compute hardware are the same. However, with the advancement in heterogeneous hardware systems, such as different generations of CPUs and GPUs, it may not be practical to have this assumption. For example, a tuner trained for a

DL workload on a CPU may not generate efficient tensor programs on GPUs. Transfer learning has been advantageous in mitigating these limitations. It can learn the context from the neural network tasks and apply them to the new context.

Hence, the need for a transfer learning-based methodology requiring lesser data and quick adaptability to the new hardware is paramount. The cost models trained on limited hardware or neural network tasks lack transfer learning capabilities. Consequently, it is efficient to map the heterogeneous feature space across devices and fine-tune only the required features applicable instead of retraining from scratch. Where hand-tuned libraries fall short of providing optimized support to new hardware and operators, auto tuners requiring lesser data to learn can reduce the tuning time and the time required for online device measurement by focusing on learning high-performance kernels. Recent works have proposed transfer learning methods for the same source and target hardware [6, 12, 22]. We suggest heterogeneous transfer learning by mapping the kernel as a feature space in a new context.

This paper analyzes the current methods used to generate tensor programs using transfer learning for CPU and GPU-based systems. We conduct probabilistic and exploratory analyses to achieve comparable results using less data than the baseline across various split strategies. We propose a transfer learning approach to generate efficient tensor programs with less tuning time and fewer kernel measurements across heterogeneous hardware. The significant contributions of this paper are as follows:

- perform extensive study on the existing work to extract and learn from the joint significant neural network and hardware features
- define the proposed methodology based on fewer kernels measurements and efficient transfer learning
- implement an optimized tuner based on the key points above and present results for heterogeneous transfer learning. We could achieve on-par or better mean inference time with 3x-5x improved tuning time and up to 47% reduced dataset.

The remainder of the paper is organized as follows: Section 2 gives the requisite background to understand the problem and presents the related work in this area. Section 3 describes the proposed methodology used in the study. Sections 4 and 5 discuss experimentation and results, respectively. Section 6 provides a conclusion and potential future steps.

2 BACKGROUND AND RELATED WORK

2.1 Cross-device Learning

The search space in cross-device learning is enormous, ranging from the orders of millions (CPU) to billions (GPU). It leads to ample search space and incurs high costs in terms of search time. Transferring auto-schedule knowledge from pre-tuned kernels to untuned kernels can play a significant role. As discussed here, researchers have proposed solutions to address various aspects of transfer learning [6]. In [6, 17], authors have classified tasks into classes to have a better selection of optimizations. [8] uses a hierarchical LSTM–based approach to predict throughput based on the opcodes and operands of instructions in a basic block. The authors propose a solution that is easily portable across various processor micro-architectures. Other research uses a cost model query optimizer to improve resource utilization and lower operational costs [14]. In [20], the authors propose an end-to-end pipeline to optimize synchronization strategies given model structures and resource specifications, lowering the bar for data-parallel distributed ML across devices. Another work [1] employs reinforcement learning to develop an adaptive sampling algorithm to alleviate costly hardware measurement. Our work proposes efficient pruning of datasets to learn joint kernel and hardware features. It enables efficient tuning by considering prominent kernels.

2.2 Machine Learning-based Auto Tuners

ML-based approach to automatically optimize tensor programs is a heavily researched area [4, 5, 7] focusing on tensor compilers and DL workloads [18]. [21] is a framework to generate tensor programs for the DL workloads. It explores optimization combinations by sampling programs from a hierarchical search space representation and optimizes multiple sub-graphs using a task scheduler. [15] employs LSTM to formulate the scheduling process as a sequence of optimization choices. Furthermore, [19] described an approach for automatically generating and optimizing numerical software for processors with deep memory hierarchies and pipelined functional units. Such an approach makes its adoption across server and mobile platforms feasible.

Additionally, authors have employed reinforcement learning (RL) in various works. In [9], the authors explore the feasibility of formulating the integer linear programming solver into a graph neural network-based policy to auto-generate a vectorization scheme. Further, domain-specific compilers like COMET [10], JAX [2], and NWChem [16] are actively researched to optimize the underlying program's execution. [10] performs domain-specific, hardware-agnostic optimizations that rely on high-level semantic information and are applied at high-level IRs. Lately, in [12], authors have proposed a one-shot tuner for the tensor compilers. Its limitation is that it considers only task-specific information, which prevents it from being applied for transfer learning. On the other hand, we employ neural network and hardware platform information so that the auto-tuner can learn better.

3 DESIGN AND IMPLEMENTATION

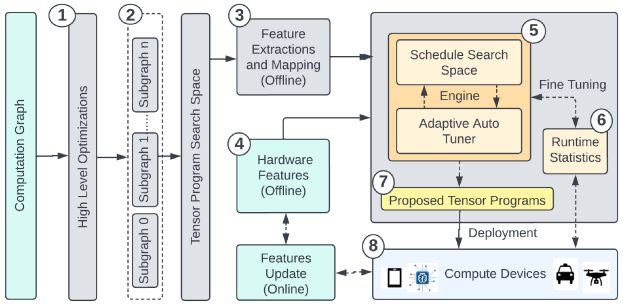


Figure 1: Overview of the Proposed Framework

This section unfolds our methodology as follows: Section 3.1 provides an overview of the entire flow and individual components,

Section 3.2 elaborates on the proposed optimizations and opportunities for efficient heterogeneous transfer learning with less dataset, and Section 3.3 explains the adaptive auto-tuner architecture.

3.1 An Overview

Figure 1 provides an end-to-end flow of our framework. We used TVM `v0.8dev0` as a base to present the heterogeneous transfer learning. The user leverages the TVM API to (1) provide the computation framework in the supported format (TensorFlow, PyTorch, etc.). It further (2) undergoes high-level optimization and subgraph partitioning to generate subgraphs of a smaller order. The induced subgraphs form the search space for the feature extractions. Additionally, as a feature set, (3) domain-specific information is extracted from these subgraphs, like kernel dimensions, tensor operations, etc. For each data-point entry or kernel, (4) we store the hardware information like hardware architecture, maximum number of threads, registers and threads per block, and shared memory. This forms part of the static hardware dataset. We perform a probabilistic and exploratory study on this feature set to identify the features of high importance. The hardware characterization assists in mapping the features from source to target in case of a different hardware than the source. We (5) train auto-tuner on this dataset by extending a one-shot tuner [12]. Once the auto-tuner is trained, it can be used to perform auto-generation of tensor programs on the target device with or without re-training. If the user wants to fine-tune the auto tuner, we provide a capability to (6) fine-tune the auto tuner using online hardware features. We have experimented with selective tasks retraining, and are still working on a methodology for selective feature training. The non-parametric approach reduces the retraining time and size of the dataset required. Also, we used attention heads to support memory-augmented fine-tuning using bidirectional LSTM. Since the training strategy is based on the one-shot paradigm, the baseline auto-tuner model undergoes a one-time complete training phase alleviating the large data requirement on the target device for full retraining. Hence (7) proposed tensor programs are (8) deployed on the target hardware and their performance is evaluated.

We used the TenSet dataset [22] for the baseline measurements. As per authors claims, the dataset can be leveraged for generating the tensor programs using transfer learning. The basis is diversity in the dataset leading to generalization and multi-platform performance data points (measured on CPUs of Intel, AMD and ARM and NVIDIA GPUs). Section 3.3 discusses the applicability of these bases for efficient transfer learning using the proposed auto-tuner. For more information on the dataset, one can refer to [22]. Table 1 summarizes the hardware and associated characteristics that we have considered in our study.

3.2 Hardware-Aware Kernel Sampling

We have extensively studied and conducted an experimental examination of the TenSet dataset to understand the neural network and hardware features that can influence the metrics like flops, throughput, and latency. The dataset consists of over 13,000 tasks from 120 networks measured on six hardware platforms for the metrics like throughput and latency under different neural network and hardware parameters resulting in over 52 million measurements. As shown in Table 1, we have considered the first four platforms from the dataset to learn the task, schedules applied to them, and associated performance along with the hardware parameters. The latter half of the hardware is used to evaluate and establish the usefulness of cross-device or transfer learning. In addition to the neural network information like tensor operation and input and output shape, we have also considered hardware parameters, as shown in Table 2. Here, we have presented hardware features only for CPU and GPU but it can be extended to other heterogeneous devices. Analyzing the dataset, we identified that most high-performant kernels are associated with specific hardware parameters. Such probabilistic sampling also mitigates skewed kernel selection, avoiding kernels that can lead to lower FLOPs or invalid computation graphs. We removed the measurements and kernels which were invalid or low-performing. In the existing cost modeling, it is observed that the large search space selected via random sampling of kernels causes performance regression.

Table 1: Compute Hardware Description

Hardware Platform	Processor	Remarks
Intel Platinum 8272CL @ 2.60GHz	CPU	16 cores, AVX-512
AMD EPYC 7452 @ 2.35GHz	CPU	4 cores, AVX-2
ARM Graviton2	CPU	16 cores, Neon
NVIDIA Tesla T4	GPU	Turing Architecture
NVIDIA GeForce RTX 2080	GPU	Turing Architecture
NVIDIA A100	GPU	Ampere Architecture
NVIDIA A40	GPU	Ampere Architecture
Intel Gold 5115 @ 2.40GHz	CPU	40 cores, Xeon

Table 2: Hardware Parameters Considered While Training

Hardware Parameter	Definition	Hardware Class	Value (bytes)
cache_line_bytes	chunks of memory handled by the cache	CPU; GPU	64
max_local_memory_per_block	maximum local memory per block in bytes	GPU	2147483647
max_shared_memory_per_block	maximum shared memory per block in bytes	GPU	49152
max_threads_per_block	maximum number of threads per block	GPU	1024
max_vthread_extent	maximum extent of virtual threading	GPU	8
num_cores	number of cores in the compute hardware	CPU	24
vector_unit_bytes	width of vector units in bytes	CPU; GPU	64; 16
warp_size	thread numbers of a warp	GPU	32

We pruned the vast search space for the tensor program generation. The search task based on random sampled kernels' measurements is unreliable when the search space is not rich. We observed that specific hardware parameters, like the number of cores, are less valuable features than flop count regarding latency. It is mainly because of the need for diversified kernel combinations and hardware features in the considered dataset. Hence, we evaluated the combination of FLOPs count, kernel shape, and execution time on a given hardware for various tensor operations. As shown in Table 3, we observed that the CPU and GPU behave similarly when selecting the best shape for a kernel. Although, the execution time differs significantly based on the compute hardware. Hence, we sampled the kernels based on joint exploration of tensor operations, kernel shapes, and hardware parameters based on the FLOPs count and execution time. The extracted initial kernels are from six diversified

Table 3: Neural-Network And Hardware Features Characterization

Sampled Kernels	#Kernel_Shapes		Max GFLOPs		Tensor Shape	Mean Execution Time (ms)			
	CPU	GPU	CPU	GPU		EPYC-7452	Graviton2	Platinum-8272	T4
T_add	229	388	8.59	8.59	[4, 256, 1024]	180.97	81.25	92.86	4.31
Conv2dOutput	60	27	1.20	1.07	[4, 64, 64, 32]	40.94	14.21	19.11	2.07
T_divide	24	69	0.003	0.003	[8, 1, 1, 960]	0.07	0.05	0.11	0.10
T_fast_tanh	9	9	0.008	0.008	[4, 1024]	0.43	0.43	0.53	0.97
T_multiply	105	150	8.92	8.92	[4, 256, 4096]	320.74	48.08	95.65	0.55
T_relu	300	1257	73.46	73.46	[4, 144, 72, 8, 64]	0.52	5.70	0.72	0.23
T_softmax_norm	27	27	0.016	0.016	[4, 16, 256, 256]	1.01	2.78	4.08	0.19
T_tanh	9	9	0.905	0.629	[8, 96, 96, 3]	5.55	33.48	50.55	0.16
conv2d_winograd	0	33	NA	0.868	NA	NA	NA	NA	0.93

*CPU: EPYC-7452, Graviton2, Platinum-8272; *GPU: T4; *NA: operator is not present in the considered CPU dataset **Measured on Nvidia GeForce RTX 2080

neural networks to encompass prominent classes. Still, it will be an ever-growing list, and we are working on an intelligent algorithm to achieve the same whenever introducing a new network class. Experimenting with `rmse` and `ranking loss`, we established that `rmse` performs better based on inference time. Hence, we chose `rmse` for our tuner's performance evaluation. We learned from the measurement records' costs of the selected kernels that it contains local minima. To avoid local minima, we leveraged simulated annealing implemented in TVM. But it is computationally heavy and takes significant time to find global optima. We aim to optimize it in our future work.

3.3 Auto-tuner Architecture

We have extended the one-shot tuner to experiment with a joint neural network and hardware features as part of the task. Our tuner consists of bidirectional LSTM and attention head to learn from the sequential data. After testing multiple configurations, we chose the near-optimal values based on empirical evaluation, including a batch size of 16, 200 epochs, and other hyperparameters listed in Table 4.

Table 4: Auto-Tuner's Hyperparameters

Hyperparameter	Value
Batch	1, 4, **16**, 64, 256
Epoch	100, **200**, 400
Learning Rate	1e-3
#Bidirectional LSTM Layers	3
Attention Head (fine tuning)	2
#Unrolling Steps for Attention Head	2
Optimizer	Adam

4 EVALUATION

4.1 Experimental Setup

Platform: We used heterogeneous architectures like Nvidia GPUs-RTX 2080, A100, A40, and Intel Xeon CPU for our study experiment. We chose different generations of NVIDIA GPUs to study the impact of architectural differences.

Dataset and Model: This work use TVM v0.8dev0 and PyTorch for implementations. We used XGBoost (XGB), multi-layer perception (MLP), and LightGBM (LGBM) based tuners as a baseline.

Baseline: For the baseline, we used the TenSet dataset, commit 35774ed. Based on the previous work [22], we have considered 800 tasks with 400 measurements as the baseline. We used Platinum-8272 for the CPU dataset and Nvidia Tesla T4 for the GPU dataset.

4.2 Dataset Sampling

As discussed in Section 3.2, we have sampled the dataset. By employing the data sampling strategies based on the feature's importance in terms of FLOPs count, we could reduce the GPU dataset by 43% and the CPU dataset by 47%. As shown in Table 5, we could gain the training time overall.

While sampling the dataset, it is essential to ensure the accuracy of the resultant cost model or tuner trained over the sampled dataset. Hence, we compared the top-1 and top-5 accuracy with the pairwise comparison accuracy (PCA). To explain briefly, if y and $\hat{y}$ are actual and predicted labels, then we calculate the number of correct pairs, CP, by performing elementwise xor, followed by elementwise not on y and $\hat{y}$. Further, we take the sum of the upper triangular matrix of the resultant matrix. The PCA is calculated then using equation 1.

$$PCA = CP/(n * (n-1)/2); n = len(\hat{y}) \quad (1)$$

The cost models trained on the baseline and sampled dataset performed on par.

To have a fair comparison we have trained XGB, MLP, and LGBM tuners on the baseline and sampled data using three split strategies as explained here:

- *within_task*: the dataset is partitioned into train and test based on the measurement record. Once the features are extracted for each task, it is shuffled and randomly partitioned.
- *by_task*: a learning task is used to partition the dataset randomly based on the features of the learning task.
- *by_target*: partitioning is performed based on the hardware parameters

To avoid skewed sampling, tasks with too few measurements were excluded. Further, we have considered the tasks based on the FLOPs of tensor operations occurrence probability as shown in Table3. The latency and throughput of these tasks are recorded by executing them on the computing hardware. Table 5 presents the gain in

Table 5: Reduction In Dataset Size And Train-time (by split strategies)

Target Hardware	Dataset	Size	XGBoost (train-time (sec))			MLP (train-time (sec))			LightGBM (train-time (sec))		
			within_task	by_task	by_target	within_task	by_task	by_target	within_task	by_task	by_target
GPU	Baseline	16G	1504	1440	454	3000	2434	3150	1574	780	4680
	Sampled	9G	1406	1169	339	1968	1655	2464	1175	595	3637
CPU	Baseline	11G	1490	1265	428	3143	2623	2043	1131	636	3946
	Sampled	6.8G	905	780	354	2091	1672	1270	489	387	2435

time-to-train for the sampled dataset. It can be seen that in the case of CPUs, the gain is up to 56% for LGBM while using *within_task* split strategy during training. GPUs also have shown an increase of up to 32%.

4.3 Tensor Program Tuning

Here, we discuss metrics to show the effectiveness of our proposed approach over the baseline.

Table 6: Pairwise Comparison Accuracy

Hardware	Dataset	Split Scheme		
		By Target	Within Task	By Task
NVIDIA A100 GPU	Baseline	0.8476	0.8931	0.8565
	Sampled	0.844	0.8953	0.8534
Intel Xeon CPU	Baseline	0.8477	0.8506	0.8651
	Sampled	0.8434	0.8456	0.861

Table 6 shows the Pairwise Comparison Accuracy(PCA) earlier discussed in Equation 1 for each split scheme for our sampled dataset with baseline dataset for NVIDIA A100 GPU and Intel Xenon CPU. We observe that accuracy hardly changes with the sampled dataset. We also observe a similar trend for other architectures used in this study.

Table 7: Inference Time Comparison (Seconds)

Target Hardware	Baseline Dataset		Sampled Dataset	
	W/o Transfer Tuning	W/ Transfer Tuning	W/o Transfer Tuning	W/ Transfer Tuning
A100 GPU	578	391	585	400
A40 GPU	627	416	599	175
RTX2020 GPU	18.67	27.68	17.37	841.74
Xeon CPU	91.34	282.2	85.22	189.25

Table 7 shows the inference times for baseline and sampled datasets with and without transfer tuning for NVIDIA A100, A40, RTX 2020 GPUs, and Intel Xeon CPU. We observe significant advantages with the sampled dataset as the inference times are much lower than the baseline dataset. The numbers discussed here are for the XGBoost tuner. Please refer to our GitHub[1] for inference numbers using multi-layer perception (MLP) and LightGBM (LGBM) based tuners as well inference numbers for various batch sizes (1,2,4,8) and detailed logs for different architectures used in this study. We observe similar trends advantageous to the sampled dataset.

[1]https://github.com/xintin/TransferLearn_HetFeat_TenProgGen

4.4 Evaluation of Heterogeneous Transfer Learning

Different transformations can be applied for a given compute graph consisting of tensor operation and input and output tensor shapes, varying their performance on target hardware. For example, in a conv2D tensor operation, tiling is dictated by whether the target is GPU or CPU because of grid and block size bound in GPUs. A large tiling size that may be valid in CPU may be invalid in GPU. Also, not all combinations are performant. We learned the joint-optimized schedules for a kernel and hardware using the TVM auto-scheduler. Then, we applied it to similar untuned kernels using the attention mechanism. To make it efficient, we sort the kernels as per their occurrences and total contribution to the FLOPs count. Then we tune the selected few significant tensor operations.

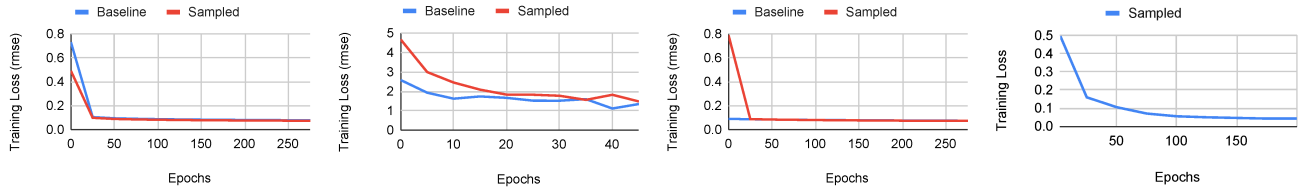
We have evaluated our methodology using three architecturally different networks on CPU and GPU. Contrasting to the baseline, where the tasks are fetched randomly tuned, we select the tasks contributing more to the FLOPs. As shown in Table 8, we could achieve the on-par mean inference time with significantly reduced tuning time. On CPU, for ResNet_50, we could achieve 30%, MobileNet_50 70%, and Inception_v3 90% reduction in time on CPU. ResNet_50 suffered performance regression due to a lack of matching kernel shapes in the trained dataset for the given hardware. Whereas on GPU, we could achieve 80%-90% across all the networks. Here, we used the trainer tuned on features from neural networks and hardware.

Table 8: Evaluation Of Proposed Tuner

Target Hardware	Network	W/o Transfer Tuning		W/ Transfer Tuning	
		Time-to-Tune	Mean Inf. Time	Time-to-Tune	Mean Inf. Time
CPU	ResNet_50	128	11.93	86	12.12
	MobileNet_v3	236	5.48	71	5.57
	Incpetion_v3	614	75.27	61	73.80
GPU	ResNet_50	817	3.79	226	3.78
	MobileNet_v3	1092	1.72	136	1.75
	Incpetion_v3	2510	28.72	191	28.73

*CPU: Intel Xeon; *GPU: A100; *w/o: without; w/: with; tune time (sec); inf time (ms)

We have also compared the tuners for the epochs required to converge on the baseline and sampled data. As per the design of TVM's auto-scheduler, if they are executed for a large number of trials, evidently, the tuners, like XGB and MLP, will converge. To have a fair comparison, we have compared them by epochs. As shown in Figure 2, there is not much difference for XGB, MLP, and LGBM tuners on either dataset. On the other hand, our attention-inspired tuner performed much better by converging in a similar number of epochs but achieving twice the better error loss. The



(a) XGBoost (b) MLP (c) LightGBM (d) Our Optimized Tuner

*experimented on Nvidia RTX 2080

Figure 2: Comparing Training Convergence of Tuners

rmse for our optimized tuner is 0.04 after 200 epochs compared to 0.08 and 0.09 of XGB and LGBM, respectively. However, we are addressing an offline training overhead involved here as part of our next steps. Additionally, this is our first step, and we are also researching the instability of the tuners.

5 CONCLUSION AND FUTURE DIRECTIONS

In this research, we have demonstrated the effectiveness of the neural network and hardware parameters-aware sampling in automating tensor program generation for search-based tensor compilers. We showed the impact of various split strategies on the end-to-end optimization duration and early convergence. Mapping tensor operators to specific hardware may be crucial in a heterogeneous environment. Here, we have integrated hardware features into the evolutionary search procedure for efficient tensor program generation. We concluded that a heterogeneous features-aware training strategy could reduce training overhead regarding dataset requirements and yield effective transfer learning with fewer online measurements. After presenting our preliminary results, we intend to research selective feature training during transfer learning. Our future work includes improving the efficiency of cross-device and inter-subgraph learning with an evaluation of a scientific application.

ACKNOWLEDGMENTS

This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is also based upon work supported by the National Science Foundation under grant no. CCF-2113996. This research used resources of the Argonne Leadership Computing Facility (ALCF), which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357

REFERENCES

[1] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *arXiv preprint arXiv:2001.08743* (2020).

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/google/jax

[3] 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.

[4] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).

[5] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 219–232.

[6] Perry Gibson and José Cano. 2022. Transfer-Tuning: Reusing Auto-Schedules for Efficient Tensor Program Code Generation. In *31st International Conference on Parallel Architectures and Compilation Techniques (PACT). Chicago.*

[7] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A learned performance model for tensor processing units. *Proceedings of Machine Learning and Systems* 3 (2021), 387–400.

[8] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on machine learning*. PMLR, 4505–4515.

[9] Charith Mendis, Cambridge Yang, Yewen Pu, Dr Amarasinghe, Michael Carbin, et al. 2019. Compiler auto-vectorization with imitation learning. *Advances in Neural Information Processing Systems* 32 (2019).

[10] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. 2022. Comet: A domain-specific compilation of high-performance computational chemistry. In *Languages and Compilers for Parallel Computing: 33rd International Workshop, LCPC 2020, Virtual Event, October 14-16, 2020, Revised Selected Papers*. Springer, 87–103.

[11] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[12] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. 2022. One-shot tuner for deep learning compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 89–103.

[13] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[14] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.

[15] 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), 323–334.

[16] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus Johannes Jacobus Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Aprà, Theresa L Windus, et al. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477–1489.

[17] Gaurav Verma, Swetang Finviya, Abid M Malik, Murali Emani, and Barbara Chapman. 2022. Towards neural architecture-aware exploration of compiler optimizations in a deep learning {graph} compiler. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*. 244–250.

[18] Gaurav Verma, Yashi Gupta, Abid M Malik, and Barbara Chapman. 2021. Performance evaluation of deep learning compilers for edge inference. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 858–865.

[19] R Clinton Whaley and Jack J Dongarra. 1998. Automatically tuned linear algebra software. In *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 38–38.

[20] Hao Zhang, Yuan Li, Zhijie Deng, Xiaodan Liang, Lawrence Carin, and Eric Xing. 2020. Autosync: Learning to synchronize for data-parallel distributed deep learning. *Advances in Neural Information Processing Systems* 33 (2020), 906–917.

[21] 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. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

[22] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.