Optimizing AI Scheduling on GPUs with Ansor, Transfer Tuning, and Droplet Search

1st William Parker
Charles W. Davidson College of Engineering
San José State University
San Jose, CA, USA
william.j.parker@sjsu.edu

2nd Volodymyr Makarenko
Charles W. Davidson College of Engineering
San José State University
San Jose, CA, USA
volodymyr.makarenko@sjsu.edu

3rd Tarun Sanjeev Banala
Charles W. Davidson College of Engineering
San José State University
San Jose, CA, USA
tarunsanjeev.banala@sjsu.edu

***Abstract*—To interact with a GPU to run tasks such as AI models, the task need to be scheduled and coordinated with the rest of the device through the task scheduler. Libraries have been made that have hand tuned the kernel to run these models, but this tuning only exists for the set of GPUs the library supports and may not optimize novel operations. To solve this, work has been done to automatically search for optimizations for the kernel for any hardware and model combination. This work proposes combining two existing techniques, Droplet Search, and Transfer Tuning, on top of the Ansor task scheduler generator, to improve the speed at which the search finds optimizations. We find that combining these techniques can lead to improvements compared to when Transfer Tuning is used by itself with Ansor by up to 10%. The source code is available at https://github.com/Jaspann/transfer-tuning.**

***Index Terms*—Task Schedulers, Artificial Intelligence, Deep Learning, GPUs.**

I. Introduction

Individuals, academics, and industry all run different AI models on their devices. AI has become a part of many digital systems, and very in capabilities, such as natural language processing, image generation, object detection, facial recognition, and much more. Each model has different requirements to which it must devote resources, and each hardware component that the model may run on has different specifications. Optimizations that work on one GPU may not work on another, and due to the large verity of GPU models and options in building deep learning models, this creates a near impossible task to optimize every combination. This is especially noticeable for consumer-grade technology, where there is a large verity in hardware while end-user applications are quickly pushing small AI models to run locally rather than dealing with hosting the model centrally.

Traditional solutions like cuDNN [1] propose libraries that optimize the most important parts of the AI models on the most popular GPUs. This paradigm does not work for everyone though, especially considering the rapid pace of AI development and deployment paired with growing historical base and new options for compute. Developers using these models suffer as well, as they need to sacrifice options that in theory are a better fit for the system, but due to a lack of support cannot be implemented unless there is significant development put into these problems that are completely separate from the development of the system.

In response, techniques such as Ansor [2] from the TVM (Tensor Virtual Machine) [3] project have been developed to programmatically search for optimization techniques to apply to optimize the model with the task scheduler. Our research lies in testing combining optimization techniques in Ansor to measure how they compare apart versus when combined together.

II. Background

In this section we will discuss the systems used to preform task scheduler optimization for deep learning models.

A. cuDNN

cuDNN [1] is NVIDIA's proprietary CUDA library for deep neural networks. It provides highly optimized implementations of common deep learning operations (such as convolutions, pooling, and activation functions) through pre-written kernels specifically tuned for NVIDIA GPUs. While cuDNN works with the CUDA runtime system for scheduling optimizations such as asynchronous kernel execution and computation/-data transfer overlap, the actual task scheduling is primarily handled by CUDA itself. Being vendor-specific and closed-source, cuDNN's optimizations are limited to NVIDIA GPUs and specific hardware configurations. This creates limitations in two ways: first, the optimizations are only available for supported NVIDIA hardware, and second, the optimization strategies are pre-determined rather than adaptable to new scenarios. TVM's authors argue this approach is insufficient for the diverse range of modern deployment scenarios, where AI models need to run efficiently on a variety of hardware platforms beyond just NVIDIA GPUs.

B. Tensor Program Optimization

Halide language, developed in 2013, was one of the first solutions to optimizing stream and stencil programs [4]. The authors were among the first to consider the problem of optimizing image processing pipelines and to show an algorithm capable of finding better solutions compared to manual code optimizations.

The general problem of optimizing ML program structure is known as Tensor Program Optimization. Chen et al. have introduced TVM, an extension of Halide to the machine learning setting [3]. TVM is a static tensor program compiler.

TVM enables high-level tensor program optimization by implementing multiple optimizations for a variety of hardware configurations while supporting popular ML frameworks, bypassing the limitations of cuDNN. Higher-level optimization allows handling GPUs with different operation primitives, or different types of accelerators such as TPUs. Graph-level optimization includes **operator fusion** and **data layout transformations**. Operator fusion reduces the function call overhead by combining eligible adjacent operations together, generating a 1.2-2.0x speedup. Data layout transforms seek to optimize the data locality for a particular hardware configuration, such as calling a kernel with tiling corresponding to a particular matrix size. The challenge in operation fusion and data layout transforms lies in having access to hardware-specific implementations for each transform, which is infeasible due to a combinatorial explosion of possible configurations. The authors address this through a Halide-inspired schedule generation and search.

While TVM introduced strong statically compiled performance boosts, its results still rely heavily on pre-defined templates. Jia et al. proposed TASO - a more flexible graph search algorithm [5]. End-to-end DNN structure optimization methods such as Apollo and . With Ansor, Zheng et al. [2] improved on TVM, TASO, and Halite by introducing a more refined hierarchical search space, a learned cost function, and a search efficiency optimizing task scheduler. The search space hierarchy is composed of high-level sketches (e.g., operation fusion options) and low-level annotations (e.g., tile sizes). The algorithm samples random configurations, and performs the evolutionary search. The authors note that using a learned model for evaluations significantly improves performance. They further improve the search efficiency by allocating more resources to configurations that are estimated to have the best performance by a task scheduler.

III. Motivation

AI models are extremely complex, and running them takes significant processing. Optimizing inference is critical for deployment, and improvements at any level can save everyone time and money. To this end, we wanted to research optimizations for running models on the GPU. While looking into optimizations, we noticed the capabilities of Transfer Tuning [6] and Droplet Search [7], [8]. Both tasks help optimize the use of the GPU on the system, so it appeared to be an applicable research topic.

IV. Related Works

Further optimizing the task scheduler for deep learning tasks is a very active field of study. Transfer Tuning and Droplet Search have both been successful in their own ways. This work takes the additional step of combining these two works on top of TVM to create a solution that improves beyond using either one alone.

A. Transfer Tuning

When multiple models need to be tuned and the models have some similarities between them, Transfer Tuning can be used. The technique assumes that there are task scheduling optimizations that can be shared across the models, and that, due to searching via Ansor, some optimizations may have found on some models but not others. The technique identifies these auto-schedulers, and attempts to apply them to the other models. The authors proved that the technique can drastically optimize the models, requiring Ansor to run for 10x as long to match it's performance at scale.

This solution provides better results when more models optimized via Ansor, as this means that there are more potential optimizations that can be transferred to each other. This means that if applying the optimizations from one model is unfruitful, another model may be a better fit.

B. Droplet Search

Droplet Search also selects the best candidate after tuning with Ansor as a starting point for it's optimization. Droplet Search works similar to gradient decent, where the search space is a multi-dimensional space with each potential variable for optimization as it's own dimension. The data from the optimal Ansor candidate is set as the starting point of this space, and the search attempts to optimize in a process iteratively moving through the search space, with each step only moving in one dimension at a time. The authors prove that this method works great for hardware optimizations. Additionally, the algorithm can always find the most optimal state, as they also prove that all minimas are equivalent in their space, removing the issue of local minimas.

The search is a greedy heuristic, optimizing the best variable at the current step. It looks at all of the closest options, searching within the neighborhood, to find the best option. The search is repeated until no variable provides a solution that produces an improved optimization. The algorithm supports parallel execution, making it efficient for hardware optimization tasks.

V. Design

To implement our test, we decided to start development based on Transfer Tuning's work. The Transfer Tuning paper provides a repository with clear and in-depth usage of TVM, making it a great starting point, as we could use the code directly for our purposes and allowed us to better understand the necessary concepts like TVM. From there, on TVM's Github page found pull request number 16499, which re-added Droplet Search into TVM for Ansor. The requests

seemed to have been accepted by the authors of TVM but was not merged. We decided that fetching this branch and implementing the function would provide the best solution as it has been approved by the authors.

Other changes such as editing the tests to interact with the GPU were done. This involved adding a new device in the device_info.json folder that targets CUDA. To interact with the host device's GPU, during testing there was an additional requirment for the host OS to be Ubuntu, the CUDA Toolkit and nvidia-docker2 packages are to be installed as well. We updated how the Dockerfile.main_gpu file is ran to use the newer configuration options to allow for GPU pass-though on modern Docker installations.

Once the configurations were completed, the environment was ready to start implementing the code. We follow the first few steps of downloading the models and running Ansor as prescribed in Transfer Tuning's README. In each step, the device name is set to use our CUDA device. Once that is complete, the model and data directories are copied so that one directory can run Droplet Search before both sets run Transfer Tuning and are evaluated. Droplet Search is implemented as a function that takes in the TVM log file of the model, and from this information appends the solution it found to the end of the file. The method implements Droplet Search as defined in the paper under using TVM v0.16.

After Droplet Search has been applied to the copy of the logs, both tests run the rest of the instructions. The only exception that is made is in the last two commands. These are not run, as Transfer Tuning ends with further tuning the original models with Ansor, for a comparison of how long Ansor would take to match the performance of Transfer Tuning. As that is not the point of this work, it was not done. Instead, the results can easily be seen by looking under data/results/tt_multi_models for the Transfer Tuning with and without Droplet Search for each model combination.

VI. Evaluation

To evaluate this system, we needed to compare models that have pre-existing similarities for Transfer Tuning. These models should be similar architecturally in some way. The authors of Transfer Tuning have in the tests on their public code that AlexNet [9], GoogleNet [10], and ResNet50 [11] are used with each other, so we decided to test the same combinations. With these models, we ran Ansor, copied the data and run Droplet Search, and ran Transfer Tuning on both as described in the Design for 5000 trials per model.

From this, we saw near identical results when looking at how AlexNet and ResNet50 preformed after Transfer Tuning when comparing the set with and without Droplet Search. As shown in Figure 1, in GoogleNet, we found inference running about 10% faster when using Transfer Tuning with Droplet Search versus using Transfer Tuning alone. In the chart, the dark color represents the speed at which the model, after transfer tuning against the labeled model, was able to run the benchmark, while the brighter color shows the benchmark time when Droplet Search was additionally applied to the same model combination. Applying ResNet50 to the GoogleNet model produced near-identical results, so Droplet Search did not have much of an effect with this combination, while it appears to have had an effect when applying Transfer Tuning via AlexNet on GoogleNet.

VII. Case Studies

While building the tests, there were different solutions we needed to study to create our final result.

For our tests, we initially were also going to compare VGG-16 [12]. While the authors seemed to not have much problem using it, even when we ran Transfer Tuning with nothing changed, TVM consistently threw errors when it reached VGG-16. As a result, we needed to remove this from our final test array.

Droplet Search was incomparable with Ansor with the version of TVM that Transfer Tuning was using. TVM holds an internal state of what it has done, and uses it when interacting with logs, so generating the Ansor logs and then upgrading to TVM v0.16 to apply Droplet and going back down to TVM v0.8 for Transfer Tuning appears to be infeasible. To fix this, the TVM version needed to be upgraded from v0.8 to v0.16 inside the Docker container. This surprisingly did not cause much of a problem, as the only edits needed to upgrade were upgrading CMake and changing all references of tvm.relay.backend.compile_engine import to tvm.relay.backend.te_compiler.

While Transfer Tuning works on any device, some implementations for GPU support were not made public. This required reverse engineering the design to understand the requirements for implementing on the GPU as described in the Design section. Once GPU support was enabled, we saw small increase in GPU utilization when the tests were being preformed, followed by periods of minimal utilization where it appeared that Ansor was calculating the results.

Further testing needs to be done. The testing was only done on three models, and Ansor was only run for 5000 trials. This is not optimal for Transfer Tuning, where 20,000 trials

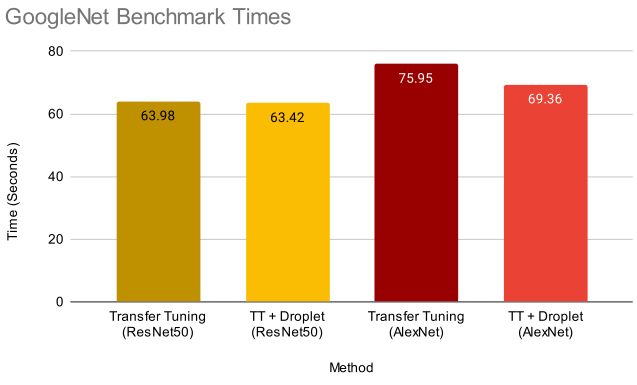

Fig. 1. Visualization of inference benchmark times on GoogleNet between using Transfer Tuning with and without Droplet Search.

are required, and better performance is possible with more devices. Testing on more devices and with different step counts on Ansor to verify the results is the next logical step for developing this work. In addition, more configurations, like trying to run Ansor in shorter bursts, like in 500-step intervals, and running Transfer Tuning and Droplet Search after each interval, may lead to promising results.

VIII. Conclusion

The rate at which AI models are being developed and utilized cannot be understated. With any large enough amount of use, even small optimizations can lead to huge boosts in productivity. Both Transfer Tuning and Droplet Search already provide additional optimizations on top of Ansor, but their combination further boosts the performance. In this paper, we show that these two algorithms can increase the speed of some models by up to 10%, providing a clear path forward for future optimizations.

References

[1] S. Chetlur, C. Woolley, P. Vandermersch, *et al.*, *cuDNN: Efficient primitives for deep learning*, Dec. 18, 2014. DOI: 10.48550/arXiv.1410.0759. arXiv: 1410.0759. [Online]. Available: http://arxiv.org/abs/1410.0759 (visited on 12/15/2024).

[2] L. Zheng, C. Jia, M. Sun, *et al.*, *Ansor: Generating high-performance tensor programs for deep learning*, Oct. 15, 2023. DOI: 10.48550/arXiv.2006.06762. arXiv: 2006.06762. [Online]. Available: http://arxiv.org/abs/2006.06762 (visited on 12/04/2024).

[3] T. Chen, T. Moreau, Z. Jiang, *et al.*, *TVM: An automated end-to-end optimizing compiler for deep learning*, Oct. 5, 2018. DOI: 10.48550/arXiv.1802.04799. arXiv: 1802.04799. [Online]. Available: http://arxiv.org/abs/1802.04799 (visited on 12/16/2024).

[4] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:5885207.

[5] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: Optimizing deep learning computation with automatic generation of graph substitutions," en, in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Huntsville Ontario Canada: ACM, Oct. 2019, pp. 47–62, ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359630. [Online]. Available: https://dl.acm.org/doi/10.1145/3341301.3359630.

[6] P. Gibson and J. Cano, *Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation*, Sep. 7, 2022. DOI: 10.48550/arXiv.2201.05587. arXiv: 2201.05587. [Online]. Available: http://arxiv.org/abs/2201.05587 (visited on 12/04/2024).

[7] M. Canesche, V. Rosário, E. Borin, and F. Quintão Pereira, "The droplet search algorithm for kernel scheduling," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, 35:1–35:28, May 21, 2024, ISSN: 1544-3566. DOI: 10.1145/3650109. [Online]. Available: https://dl.acm.org/doi/10.1145/3650109 (visited on 12/04/2024).

[8] M. Canesche, G. Verma, and F. M. Q. Pereira, *Explore as a storm, exploit as a raindrop: On the benefit of fine-tuning kernel schedulers with coordinate descent*, Jul. 15, 2024. DOI: 10.48550/arXiv.2406.20037. arXiv: 2406.20037. [Online]. Available: http://arxiv.org/abs/2406.20037 (visited on 12/15/2024).

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://papers.nips.cc/paper_files/paper/2012 / hash / c399862d3b9d6b76c8436e924a68c45b - Abstract.html (visited on 12/15/2024).

[10] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, ISSN: 1063-6919, Jun. 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. [Online]. Available: https://ieeexplore.ieee.org/document/7298594 (visited on 12/15/2024).

[11] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, Dec. 10, 2015. DOI: 10.48550/arXiv.1512.03385. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385 (visited on 12/15/2024).

[12] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, Apr. 10, 2015. DOI: 10.48550/arXiv.1409.1556. arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556 (visited on 12/15/2024).