UNIVERSITY OF LOUISVILLE

Optimizing Machine Learning Models using TVM

Joshua Hatfield, Sabur Baidya, and Basar Kutukcu

Introduction

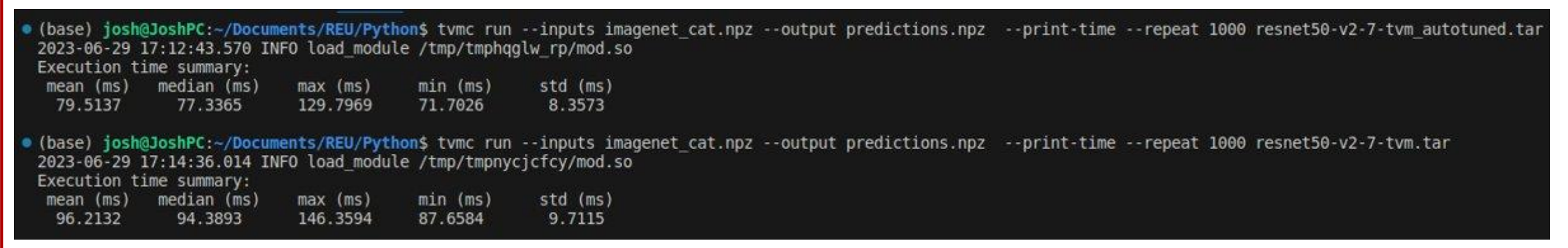
Machine learning algorithms are generally perceived in two important criteria: speed and accuracy. Thankfully, many models already can procure accurate predictions via various implementations of black box models. For the most part, we used models like that of ResNet 50 and other variations such as ResNet 18. These libraries focused on image processing models while being accurate and mainstream. On the other hand, they took some time to process. This can be an issue when looking at a live video feed that requires many images to be processed in a fraction of a second. As such, having improvements to inference latency allows the programs to be more efficient during runtime operations. Thankfully, Apache-TVM (Tensor Virtual Machine), an open-source machine learning framework which is supported in Python, helps to fill the void of machine learning optimizations by providing a plethora of tools in order to fine tune machine learning models [1].

Research Goals

For this research endeavor, we wanted to investigate how we can use TVM in order to optimize machine learning models in hopes to decrease latency as much as possible without trading off accuracy [2]. The operations in which TVM functions does not actually modify the structures in which predict the outcome of the model but change the structures in order to better suite the specific hardware. This becomes increasingly useful for embedded platforms like the Jetson TX2 and Jetson Nano which use ARM architecture. Since the TX2 and Nano both have limited computational resources due to their size [3], we feel as though looking into how these platforms can be optimized to be a useful and productive research topic [4]. Additionally, we wanted to compare how the model compares being deployed on TVM along with the same model being autotuned with tools provided by TVM. Both versions of these models would be compared in terms of power consumption and latency when specific cores are activated at key frequencies.

Methodology and Evaluation

The way in which we planned to conduct our experiments is through a linux based operating system using Ubuntu for its ease of access to the terminal and optimized kernel operations which make it a favorable experimental playground. To begin in Ubuntu, we cloned the repository of TVM and built it inside of a conda environment for simple package management [5]. This would allow us to download requires packages such as llvm in our build of TVM [6]. From here, we were able to get a preliminary experiment using an Acer Swift 3 which deployed a sample image recognition model. The model was then tuned using TVMC (Tensor Virtual Machine Compiler) and was able to sculpt the model to the hardware available on the laptop. We were able to compare the latency during the inference stage of the model in both versions. From here, we saw roughly a 20% decrease in latency when the model is tuned using TVMC in command line (Figure 1).



```
● (base) josh@JoshPC:~/Documents/REU/Python$ tvmc run --inputs imagenet_cat.npz --output predictions.npz  --print-time --repeat 1000 resnet50-v2-7-tvm_autotuned.tar
  2023-06-29 17:12:43.570 INFO load_module /tmp/tmphqglw_rp/mod.so
  Execution time summary:
   mean (ms)   median (ms)    max (ms)     min (ms)     std (ms)
    79.5137      77.3365      129.7969     71.7026      8.3573

● (base) josh@JoshPC:~/Documents/REU/Python$ tvmc run --inputs imagenet_cat.npz --output predictions.npz  --print-time --repeat 1000 resnet50-v2-7-tvm.tar
  2023-06-29 17:14:36.014 INFO load_module /tmp/tmpnycjcfcy/mod.so
  Execution time summary:
   mean (ms)   median (ms)    max (ms)     min (ms)     std (ms)
    96.2132      94.3893      146.3594     87.6584      9.7115
```

Figure 1: Running the Models in Command Line

The figure above shows the results from a model split into two separate parts: preprocess and postprocess. Preprocessing took the image and translated the picture into something the computer could understand through convolution layers that can take out key information of the image such as patterns, shapes, and other relevant information [7]. Postprocess was in control of the part that we can tangibly understand with ease: the results. In order to get these, the model made its top 5 inferences along with a confidence percentage for each. These inferences were stored in a npz file to be looked at during the experimental phase. This experiment was performed in the command line once the models were already compiled. In order to get the most accurate results, we were able to repeat the run 100 times using functionality from TVMC.

We then took an average of the modules inference times in terms of mean, median, max, min, and std (Figure 2). Additionally, we found that once this test ran multiple times, the times were decrease drastically due to the cache memory still being active from the previous test. We used the third output from running the model for both the tuned and untuned models.

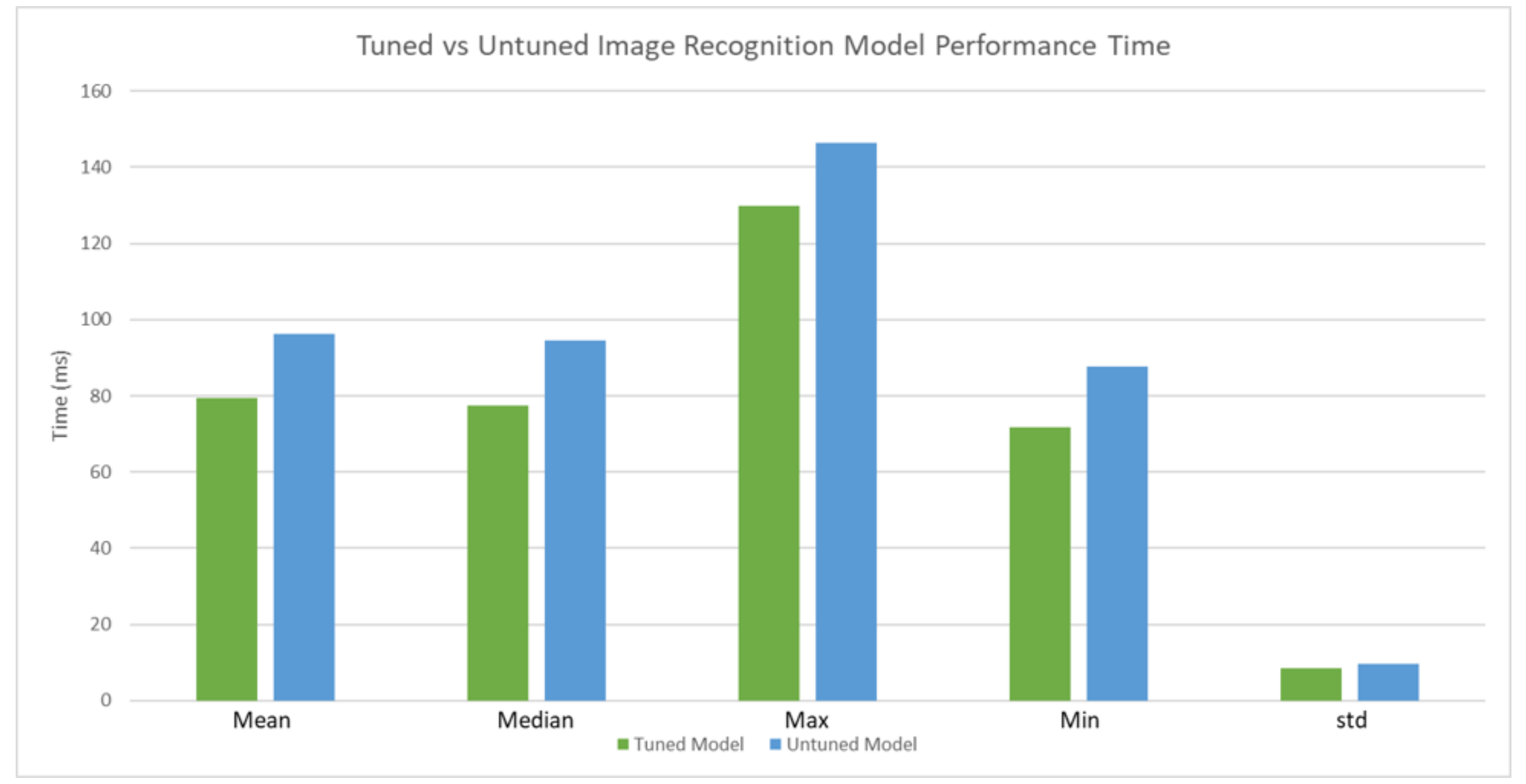
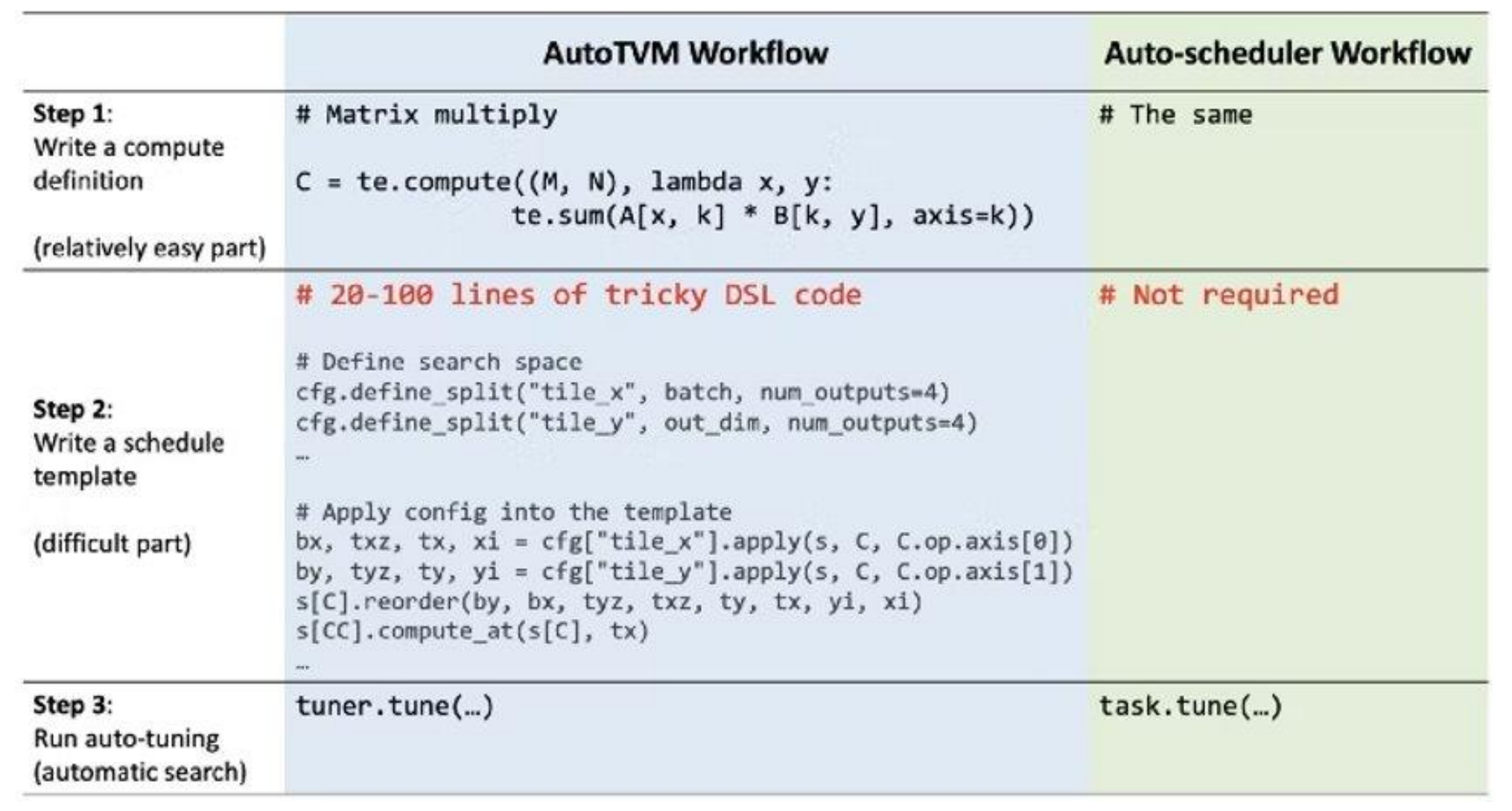


Figure 2: Tuned vs Untuned Image Recognition Performance Time

When it comes to optimization, there are only two viable options in TVM: AutoTVM and Auto-Scheduler. These two have similarities and different seen in Figure 3. The biggest takeaway from the two is that the workflow for Auto-scheduler is much easier to work with. Additionally, Auto-TVM offers a greater benefit in terms of performance [8]. On the other hand, AutoTVM does allow the user to fine tune more parameters to see which of the knobs play a significant role in terms of optimization.



	AutoTVM Workflow	Auto-scheduler Workflow

Step 1:
Write a compute definition
 	
(relatively easy part)
	`# Matrix multiply`
	
	`C = te.compute((M, N), lambda x, y:`
	`        te.sum(A[x, k] * B[k, y], axis=k))`
		`# The same`

Step 2:
Write a schedule template
 	
(difficult part)
	`# 20-100 lines of tricky DSL code`
	
	`# Define search space`
	`cfg.define_split("tile_x", batch, num_outputs=4)`
	`cfg.define_split("tile_y", out_dim, num_outputs=4)`
	`…`
	
	`# Apply config into the template`
	`bx, txz, tx, xi = cfg["tile_x"].apply(s, C, C.op.axis[0])`
	`by, tyz, ty, yi = cfg["tile_y"].apply(s, C, C.op.axis[1])`
	`s[C].reorder(by, bx, tyz, txz, ty, tx, yi, xi)`
	`s[CC].compute_at(s[C], tx)`
	`…`
		`# Not required`

Step 3: Run auto-tuning (automatic search)	`tuner.tune(…)`	`task.tune(…)`

Figure 3: AutoTVM vs Auto-Scheduler

Conclusions and Future Work

We were able to determine that TVM provides a variety of tools that are helpful in reducing inference time in machine learning models. The usefulness of TVM is best used during runtime operations in which the models are responsible for looking at a stream of images that could be used for something like that of an object detection algorithm in a drone. TVM use cases are extremely useful for the future and make the prospects of optimizing machine learning exciting. For the future, we would like to investigate how machine learning is affected on embedded platforms like that of the Jetson Nano and Jetson TX2. Both systems have limited amounts of processing power due to their small sizes. As such, fine tuning operations that involve machine learning become a must. Specifically, we are looking into how running operations at desired frequencies can affect things such as latency and power consumption. We expect there to be a trade off and would be happy to share these results in a report once the results are acquired.

Acknowledgement

This research was supported by the U.S. National Science Foundation (NSF) under grant CNS-2050925.

References

1. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018, October 5). TVM: An automated end-to-end optimizing compiler for Deep Learning. arXiv.org. https://arxiv.org/abs/1802.04799
2. Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2019, January 8). Learning to optimize tensor programs. arXiv.org. https://arxiv.org/abs/1805.08166
3. Haigh, K. Z., Mackay, A. M., Cook, M. R., & Lin, L. G. (n.d.). Machine learning for embedded systems: A case study. https://www.cs.cmu.edu/afs/cs/user/khaigh/www/papers/2015-HaighTechReport-Embedded.pdf
4. Nadeski, M. (n.d.). Bringing deep learning to embedded systems. https://www.ti.com/lit/wp/sway020a/sway020a.pdf
5. Miniconda. Miniconda - conda documentation. (n.d.). https://docs.conda.io/en/latest/miniconda.html
6. Installing TVM. Installing TVM - tvm 0.14.dev0 documentation. (n.d.). https://tvm.apache.org/docs/install/index.html
7. CS231n Convolutional Neural Networks for Visual Recognition. CS231N convolutional neural networks for visual recognition. (n.d.). https://cs231n.github.io/convolutional-networks/
8. Zheng, L., Yu, C. H., Wu, Z., Sun, M., & Jia, C. (2021, March 3). Introducing TVM Auto-scheduler (a.k.a. Ansor). Apache TVM. https://tvm.apache.org/2021/03/03/intro-auto-scheduler#:~:text=We%20build%20TVM%20auto%2Dscheduler,performance%20in%20a%20shorter%20time.