Optimizing Machine Learning Models Using TVM

Joshua Hatfield
College of Engineering and
Computer Science
Marshall University
Huntington, WV USA
hatfield308@marshall.edu

***Abstract*—TVM is a relatively new machine learning framework that is still being developed. It is purposed to optimize machine learning models through its autotuning capabilities and offers a wide range of tools to help developers achieve lower latency.**

Keywords*—*TVM (Tensor Virtual Machine), machine learning, optimization

I. 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. Black box models are algorithms in which inputs are received and outputs are resolved in a process that can be difficult to explain or at least these models don't reveal how they are able to reach their conclusions or results. For the most part, models like that of ResNet 50 and other variations such as ResNet 18 were used in order to produce accurate and reproducible results. These libraries focused on image processing models while being mainstream. Regardless of their ability to be consistent, a common issue could be the time it would take for these libraries to produce results. For example, 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].

II. Backstory

As a small bit of background regarding my personal history that may help put the scope of this report into a good lens, I wanted to mention that I had little to no experience with machine learning going into this REU. As a result, the first half of this experience was used familiarizing myself with the ins and outs of machine learning. A particular source of useful information came from an online course form Andrew NG on Coursera and YouTube. This is an open source of material meant to help introduce individuals to machine/deep learning [2].

III. Research Goals

For this research project, it was primarily focused on how TVM can be utilized to optimize machine learning models to see how it is able to decrease latency without causing a difference in the models' predictions [3]. The operations in which TVM functions do not actually modify the structures that are responsible for predicting the outcome of running the model. What TVM does is hone in on the physical hardware of the targeted platform and makes the model more inclined to use techniques which tailor to the general specifications of the said target hardware. This becomes especially useful when looking into the sphere of embedded systems due to their finite number of resources.

Figure 1: Jetson TX2

Respectively, the Jetson TX2 (Figure 1) and Jetson Nano (Figure 2) which use ARM architecture are good examples of these embedded platforms that may be targeted during TVM operations. Since the TX2 and Nano both have limited resources due to their size [4], looking into how these devices can be optimized to be a useful and productive research topic [5]. Additionally, comparing how the model performs generically versus how it performs being autotuned through TVMC. Both models would be compared in terms of measured power consumption and latency when cores are running at a desired frequency.

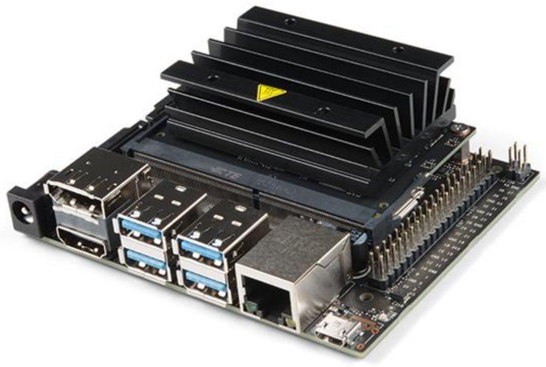


Figure 2: Jetson Nano

IV. Methodology and Evaluation

The experiments where conducted Linux command line using Ubuntu 22.04 LTS due its ease of access and optimized kernel operations. This makes Ubuntu a favorable environment for testing and deploying code. This experimental playground had its uses for building an installation of TVM for the tests to be performed. To download TVM, the repository was cloned via command line and it was then configured to allow LLVM (low level virtual machine) to be targeted which will be discussed more thoroughly later. TVM itself was built in a Conda environment which was used to support the virtual machine through a virtual environment. Conda supports simple package management which made downloading dependencies for TVM manageable [6]. TVM itself recommends using Conda for the exact purpose of managing and downloading the many dependencies on its installation documentation [7]. After installing Conda, a directory was made called tvm-build which holds all the appropriate TVM configurations and python API which would be necessary to continue.

The general setup meant to be implemented follows the steps seen in Figure 3. First, the model was to be loaded up and compiled on the target system. As a side, there is an option to either go straight into the process of compilation of the model or tuning the model to see if TVMC can fine tune parameters in the background. After the model is compiled or compiled and tuned, the model can be run and some information regarding latency can be measured using commands from TVMC.

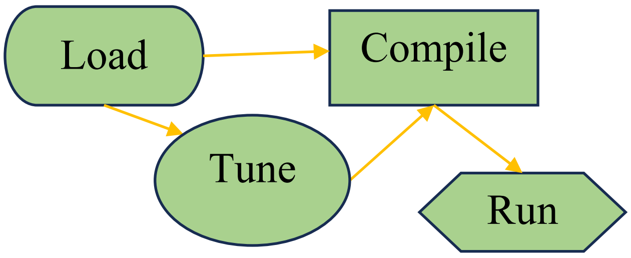


Figure 3: Example TVMC compilation flowchart

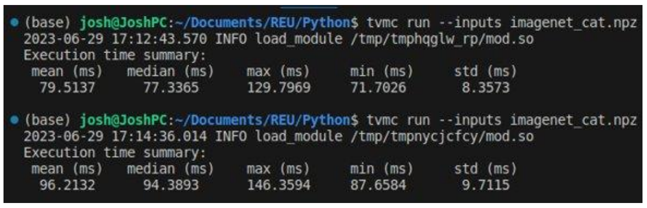
From here, a preliminary experiment using an Acer Swift 3 which deployed a sample image recognition model was performed in order to see how TVM is able to function on an image recognition model. The experiment closely followed the suggestions from TVM in "Compiling and Optimizing a Model with TVM" [8]. The model used was ResNet-50 v2 which is a pretrained library used for image recognition. The model was then compiled with LLVM as the target along with the outputs being sent to a tar file. The model included three files: mod.so (the model written in C++), mod.json (TVM Relay computational graph), and the mod.params (file holding the parameters of the pre-trained model). After compiling the model, it was ready to be used to predict and image. This would allow us to measure the time it took for the model to make this prediction. For this example, a sample image of a cat was used to test the capabilities of the model.

Figure 4: Cat source image

The processes were split up into two separate parts: the preprocess and postprocess. The preprocess example allowed the image to be transcribed into something the computer can understand. In its most raw form, a series of 1s and 0s. In doing so, the model made predictions of what the image may contain based off pattern matching and other means. Specifically, the interest was the top 5 predictions the model believed the image was. From this, it was interpreted that the model had its most confident prediction placed as the tabby cat with approximately 60% confidence.

This previous example was used without any forms of tuning mechanism and proved to be accurate in predicting the image given to it even when presented with a sample set of other images. As such, tuning the model was the next step. For this operation, using TVMC was suggested by TVM.

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. In doing so, a comparison was drawn regarding latency during the inference stage of the model in both versions. From here, there was approximately a 20% decrease in latency when the model is tuned using TVMC in command line (Figure 5). This can be attributed to TVMC autotuning features being able to target the Acer's hardware (in this case it's processor) to speed up operations.



```
● (base) josh@JoshPC:~/Documents/REU/Python$ tvmc run --inputs imagenet_cat.npz
  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
  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 5: 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 [9]. From Figure 6, this helps visualize how convolutional layers can perform the process of taking a source image and filtering it to help decipher patterns or shapes [10].

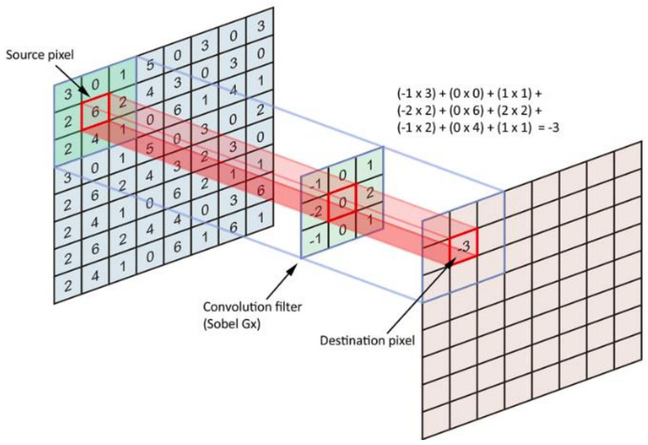

Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
(Sobel Gx)
Destination pixel
-3

Figure 6: Example Convolutional Layer

Postprocess is in control of the programming aspect that humans are able to 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, the model ran 100 times using functionality from TVMC that gathered important information such as the longest amount of time in milliseconds (ms) it took along with the shortest amount of time. Splitting the process up allowed for the model to run in a manner that it did not have to readjust the image to be in a way that the ResNet library is able to use. But performing this process separately, latency was able to be measured in a way that reflected the time it took for the model to run more accurately.

An average was gathered of the modules inference times in terms of mean, median, max, min, and std (Figure 7). Additionally, it was noticed that once this test ran multiple times, the times were decrease drastically due to the cache memory still being active from the previous test. To have a common representation among the experiments, the third iteration of the measurements was documented below.

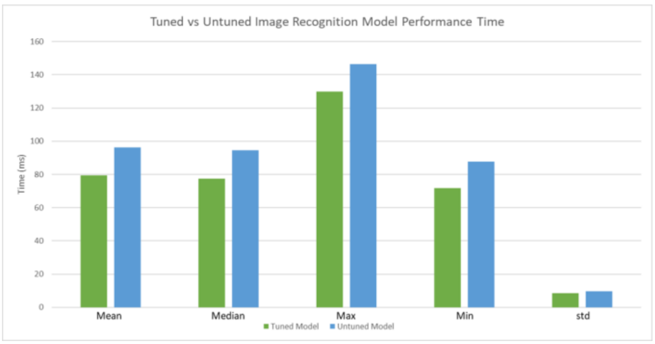


Figure 7: Tuned vs Untuned Image Recognition Performance Time

Here, the relationship between the untuned and tuned models are shown visibly in a graph produced in Excel. The green bars represent the tuned model using the CPU as the target whereas the blue bars represent the untuned model which uses LLVM as a stand in for the target. Both models performed the same task, the only key difference is how fast these predictions were formed. On average, the tuned model shows a faster ability to recognize a given image using the ResNet 50 v2 library.

Although TVM does a good job of being able to determine tuning parameters automatically, it is possible to see which parameters are being turned or switched to see similar effects via AutoTVM which will be discussed later alongside its more modern and more efficient counterpart: Auto-scheduler.

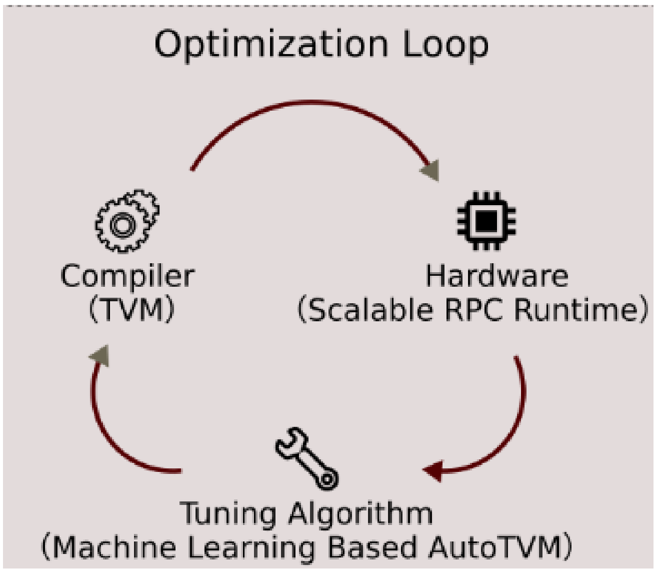
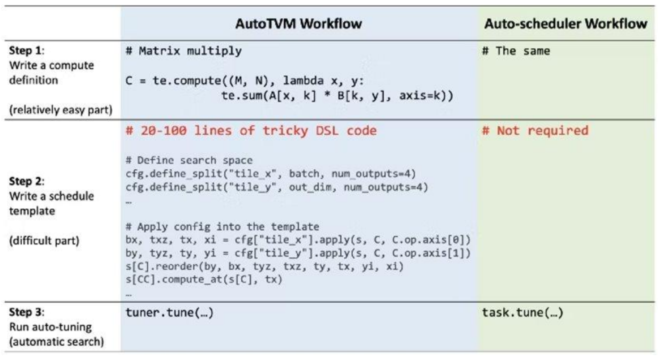


Figure 8: TVM Optimization Loop

Above (Figure 8) is a sample optimization loop that TVM utilizes many times to optimize a model. As mentioned before, TVM finds its success in targeting specific hardware and performing tasks on said hardware to find its strengths. This graph was used in an article explaining this same process [11]. TVM targets CPUs, GPUs, and other specialized accelerators to auto tune models in relatively quick succession. This iterative method allows for TVM to make small adjustments when performing tasks to ensure the best combination of optimization methods. In performing these tasks, TVM can reach a configuration of these parameters that is optimized.

When it comes to optimization, there are two reliable options in TVM aside from TVMC: AutoTVM and Auto-Scheduler. These two have similarities and different seen in Figure 9. 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 [12]. 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 9: AutoTVM vs Auto-Scheduler

V. CONCLUSIONS

All in all, it was determined 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.

Specifically, just using the auto tuner mechanism provided by TVMC shows a significant amount of improvement regarding latency without any difference to how the models are trained nor how they can predict results. A seamless application such as TVM that can provide a versatile toolkit to help reduce inference time in machine learning models illuminates the possibilities of these optimization techniques.

VI. Future Work

In the future, these findings could be applied to embedded platforms such as the Jetson Nano or Jetson TX2. Both systems have a limited number of computational resources. As such, looking into methods to optimize models they are running could be something of great benefit. More specifically, looking into how running the cores of these machines at different frequencies could affect things such as latency and power consumption is something that is being investigated. If there is a tradeoff between their ability to run at frequencies regarding both latency and power consumption, that is something which would be useful for real life runtime applications in which the Jetson products would be responsible for providing accurate and swift results consistent with what has been shown here.

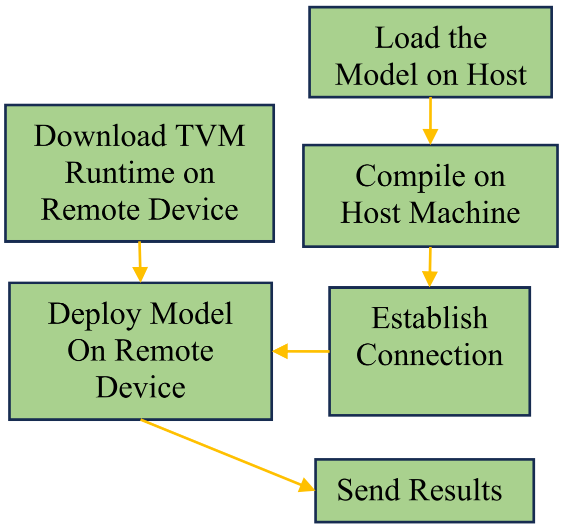


Figure 10: Flowchart depicting model deployed on remote device

Additionally, there are already resources to deploy models onto Jetson systems using TVM. They detail ways to connect to remote systems (Jetson in this case) to send models and compile onto the remote device from a host computer. This is achieved in a manner like that seen previously in this report and would be something that may have some more significance regarding TVMs ability to optimize models on specific hardware beyond that of a laptop. The only difference is that instead of TVM targeting the host CPU, it can establish a connection via an RPC server to send the models and target the remote device.

Acknowledgment

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. YouTube. (n.d.). Machine Learning Specialization by Andrew Ng. YouTube. Retrieved August 3, 2023, from https://youtube.com/playlist?list=PLkDaE6sCZn6FNC6YRfRQc_FbeQrF8BwGI&si=qwTXIB4YDszVI9AV.
3. 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
4. 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
5. Nadeski, M. (n.d.). Bringing deep learning to embedded systems. https://www.ti.com/lit/wp/sway020a/sway020a.pdf
6. Miniconda. Miniconda - conda documentation. (n.d.). https://docs.conda.io/en/latest/miniconda.html
7. Installing TVM. Installing TVM - tvm 0.14.dev0 documentation. (n.d.). https://tvm.apache.org/docs/install/index.html
8. Nunes, L., Barrett, M., & Hoge, C. (n.d.). Compiling and optimizing a model with TVMC¶. Compiling and Optimizing a Model with TVMC - tvm 0.14.dev0 documentation. https://tvm.apache.org/docs/tutorial/tvmc_command_line_driver.html
9. CS231n Convolutional Neural Networks for Visual Recognition. CS231N convolutional neural networks for visual recognition. (n.d.). https://cs231n.github.io/convolutional-networks/
10. Cornelisse, D. (2018, February 26). An intuitive guide to Convolutional Neural Networks. freeCodeCamp.org. https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/
11. Hall, S. (2021, December 21). Apache TVM: Portable machine learning across backends. The New Stack. https://thenewstack.io/apache-tvm-portable-machine-learning-across-backends/
12. 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.