DEEP NEURAL NETWORK HETEROGENEOUS COMPUTING OPTIMIZATION

A term-end report

Presented to the

Department of Computer Science and Engineering

The Chinese University of Hong Kong

In Partial Fulfillment

of the Requirements for the Degree of

Bachelor of Science

---

by

Choi Jang Hyeon

April, 2021

TABLE OF CONTENTS

1 INTRODUCTION

The objective of this research is to optimize algorithms that accelerate Deep Neural Network (DNN) heterogeneous computing. Heterogeneous computing is a technology that utilizes both the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). This brings higher performance as opposed to deploying DNN models on a CPU or GPU separately. It is worth noting that heterogenous computing is different from multi-core computing, which involves parallel calculation. Although GPUs also offer parallel computations, multi-core computation does not involve multiple architectures, while heterogenous computing consists of operating on different architectural hardware. Achieving satisfactory performance on heterogeneous hardware in spite of the architectural dissimilarities and incompatibility is a major challenge that heterogeneous computing faces [10].

Modern mainstream computers are equipped with both CPUs and GPUs. Parallel computation between these two units has been made possible through platforms such as NVIDIA's Compute Unified Device Architecture (CUDA) or the platform-independent Open Computing Language (OpenCL). As people are moving towards a more digital lifestyle, computer performance has been improving as well. Uploading videos and playing heavier computer games is becoming vastly popular among youngsters. The older generation has also been involved in technology by contributing data online, whether be it email, social networks, or online transactions. Such big data can be used to enhance overall user experience with smarter algorithms.

Deep learning can be applied to a multitude of areas including, but not limited to, facial recognition, customer behavior analysis, auto-pilot in cars, and defect detection in smart factories. SenseTime has been developing a facial recognition system that targets Chinese citizens and their technology is capable of classifying a person's gender, outfit, and age group [6]. One of the benefits of facial recognition is increased surveillance, but this is only possible after a rigorous training of a machine learning model with a substantial dataset of facial images for such categorization. Handling a large dataset could take substantial amounts of time given the subtleties of the human face. Improving and designing more efficient deep learning algorithms is crucial to drastically reduce training time.

This research project aims to improve Apache TVM, a compiler that optimizes deep neural network models to specific hardware environments. The ever-increasing variety of hardware has made it more challenging to optimally compile neural networks, and TVM helps close the gap between them.

2 BACKGROUND

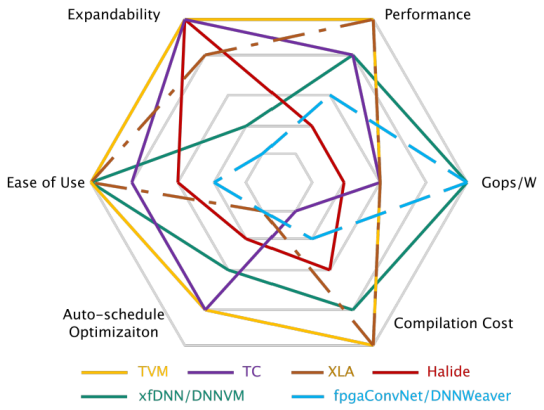
Deep learning compilers have started to gain interests as the need for deploying efficient deep learning models on different hardware grows [5]. A collaborative research done by the Beijing National Research Center for Information Science and Technology and the Center for Intelligent Connected Vehicles and Transportation of Tsinghua university have compared the

several deep learning compilers: Halide, TF(XLA), TVM, TC, DNNVM, xfDNN, and fpgaConvnet. These were benchmarked against multiple neural networks: VGG, ResNet50, ResNet52, Inception_V3, MobileNet_v1, and SqueezeNet.

	Halide	TF(XLA)		TVM		TC	DNNVM	xfDNN	fpgaConvNet*
Benchmark	CPU	CPU	GPU	CPU	GPU	GPU	FPGA	FPGA	FPGA
VGG	751.6	205	3	235	9	-	20.1	24.33	249.4
ResNet50	600.8	102	5	130	11	18.6	13.5	12.42	-
ResNet152	1683	287	13.6	343	30	48.3	36.4	34.87	153.84
Inception_v3	826.1	137	10.5	156	19	-	13.6	24.62	-
MobileNet_v1	197.3	15	6.5	50	2.9	3.17	3.4	1.5	-
SqueezeNet	139.1	10.6	7.6	20.8	4	4.4	2.5	-	-

(Figure 1. A performance benchmark of the neural network compilers represented in milliseconds. [9])

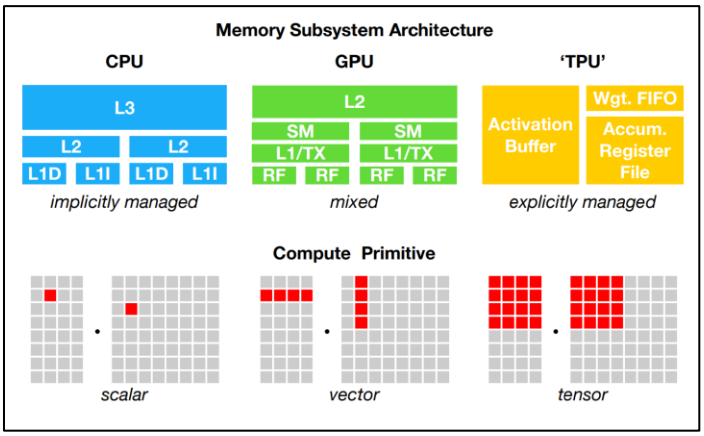
The benchmark in Figure 1 has recorded performances of the deep learning compilers. Their testing environment was conducted with an Intel Xeon CPU and NVIDIA TITAN X GPU. The table indicates that Apache TVM and Tensorflow (TF) XLA have outperformed the other compilers on both CPU and GPU while, on the other hand, Halide and fpga (field programmable gate arrays) have shown weaker results. Since DNNVM, xfDNN, and fpgaConvNet are compilers for FPGAs instead of GPUs, these compilers were not considered for research.



(Figure 2. A graphical comparison of deep learning compilers [9])

The collaborative research compared the different compilers in several aspects: expandability, ease of use, auto-schedule optimization, compilation cost, giga-operations per second per watt, and performance. At first glance, Figure 2 indicates that TC and xfDNN are the best deep learning compilers. Yet, TC is not considered to be an adequate compiler when it comes to testing performance, and xfDNN operates on a FPGA. Also, Halide turned out to be 3 to 13 times slower than TF and TVM as it was designed for image processing and uses a greedy approach in optimization instead. In terms of difficulty of use, TC and Halide require neural network models to be manually transformed, and background knowledge of FPGA and HLS tools is required when using xfDNN [9]. TVM, on the other hand, delivers a more satisfactory

performance, is easier to use, and, most of all, supports most deep learning frameworks and hardware platforms [1].

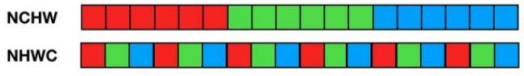


(Figure 3. Accelerators of different memory architectures and compute primitives [1].)

Figure 3 shows the architectural differences between processing units. Operator-level libraries are often specialized according to its assigned hardware and is rarely cross-compatible with other devices. TVM serves an important role in providing cross-compatibility for heterogeneous hardware; the compiler is capable of lowering a high-level specification and optimize the code for numerous hardware backends [1]. Examine the compute primitive methods more carefully, it is noticeable that the CPU computes scalar values (one-by-one data unit). GPU takes a more efficient approach and uses vectors for computation, which is a one-by-N data unit.

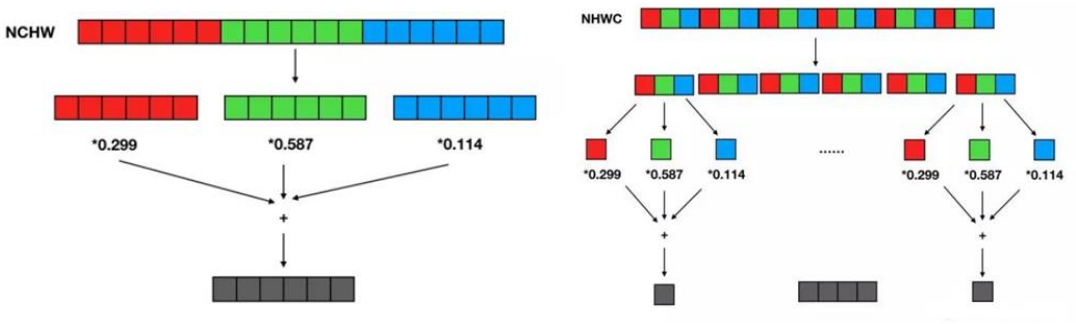
The research was conducted under the computing environment provided by the Computer Science and Engineering (CSE) department of the university. The operating system is Linux CentOS and is equipped with a high-performance 64-bit Intel Xeon CPU. The research directory was allocated with one terabyte of Big Data storage at '*/research/d1/lj2001*'. Research was continued under '*/research/d1/lj2001/TVMexamples/SEM2*' for this term.

The proposed solutions stated in the planning report includes analyzing the relationships between memory layouts, automating the optimization of the memory layout for a given DNN model, and accelerating the configuration exploration process. The memory layouts that were dealt with in this project were NCHW and NHWC, which abbreviates 'N' for the batch size, 'C' for the number of channels, 'H' for the height, and 'W' for the width of the data.



(Figure 4. NCHW vs. NHWC layouts [3])

As shown in Figure 4, the distribution of data is more dispersed in NHWC format, while it is more packed in NCHW format.

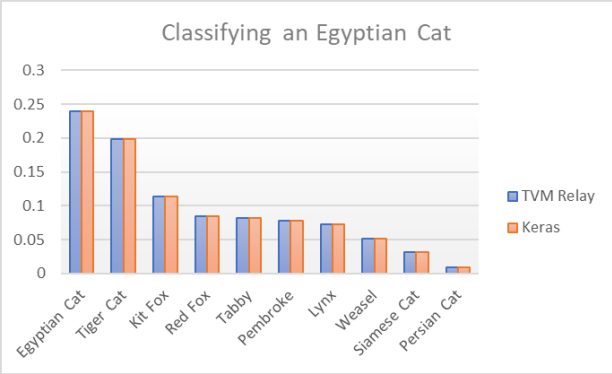


(Figure 5. NCHW and NHWC color-to-grayscale calculation [3])

Figures 5 shows how color-to-grayscale calculation is processed for both NCHW and NHWC memory layouts. It is well-known that NCHW format is more efficient for GPUs. However, it can be inferred that locality is better with the NHWC layout; the default memory layout for GPU computations is usually NCHW, and CPU-based programs use the NHWC model.

3 EXPERIMENTATION & RESULTS

After LLVM and TVM were successfully installed, the programs were tested to check if they were properly installed before proceeding with the project. The following is the result of running the tutorial template “Compile Keras Models”.



(Figure 6. Classifying an Egyptian Cat)

The two types of memory layouts have shown similar accuracies. Classifying a lion was relatively easy for the model since the traits of a lion are clear and could achieve an accuracy score of at least 97%. On the other hand, classifying an Egyptian cat was slightly more challenging since the species has a subtle difference among others, such as tiger cats, kit foxes, et cetera. The accuracy score turned out to be approximately 24% when classifying the Egyptian cat.

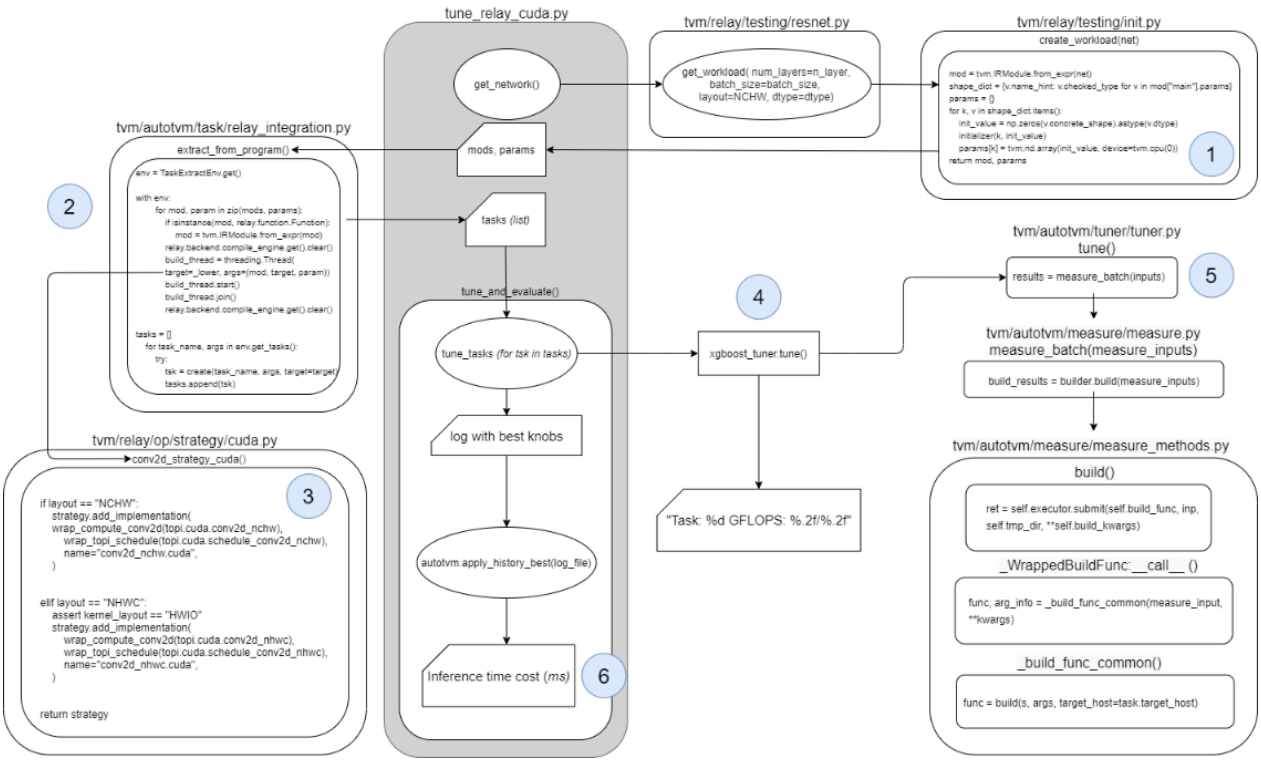
The Keras model used in this experiment was a pretrained ResNet-50, a 50-layer deep convolutional neural network (CNN) based on the ImageNet dataset that has 1000 categories,

and a network that resolves vanishing gradients [4]. The input shape of the Keras model was implemented to process data with the input shape of (224, 224, 3) with channels as the last parameter. Furthermore, images of the lion and Egyptian cat were downloaded and resized into (224, 224) to match the models' parameters. This model was then converted to a TVM Relay function via *tvm.relay.frontend.from_keras* with the default layer of NCHW and directly used to classify the images. The Relay model was then transposed using Numpy into the form [0, 2, 3, 1], sending the channel variable to the end for the NHWC format. The models then classified the images into one of the 1000 classes in Sysnet (cognitive synonyms), an interface that searches words on WordNet. The pretrained models were approximately 100 megabytes in file size and the default download directory was the Linux home path. The pretrained Keras model and example images would then be downloaded into this directory with version OpenSSL 1.0.2.

Further test were conducted with Tensorflow models to confirm whether they were compatible with the installed TVM compiler. The *from_tensorflow.py* program was run according to the "Compile Tensorflow Models" tutorial in the official TVM website. This program also makes use of ImageNet classes. It first downloads an example image of an elephant from a GitHub repository and then classified them using the Tensorflow model. After successfully running the program, results were identical to those listed on the TVM website. The output was 58% African elephant, Loxodonta Africana, 34% a tusker, 2% an Indian elephant, Elephas maximus, and 0.2% banana. At this point, there was enough confidence that the TVM compiler was successfully installed.

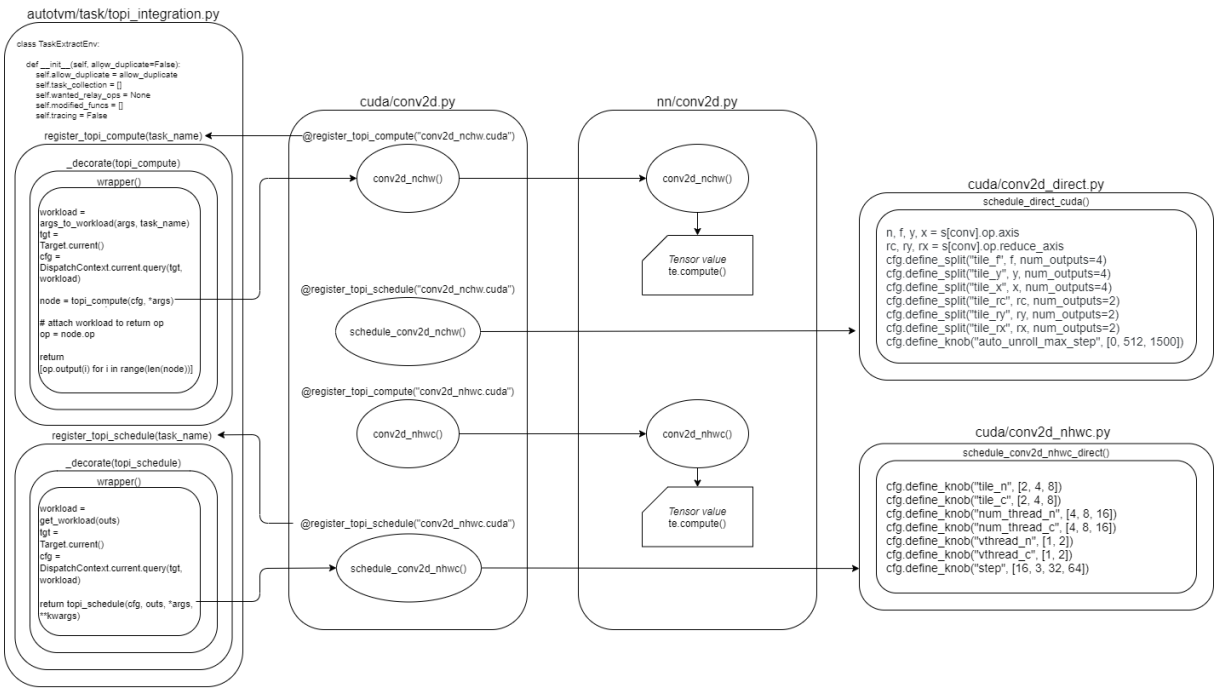
The CNN model was automatically optimized on NVIDIA GPUs with the AutoTVM package of TVM. This package automatically configures to specific devices for the best performance with the choice of several tuners: XGBoost, Genetic Algorithm (GA), Random, and Grid Search. The search-algorithm explores a design space for 24 tasks to pick out the most optimal configurations. The number of tasks could be reduced or increased by modifying the *cuda.py* file in the TVM strategy folder. Each task goes through *n* trials and measures the GPU's performance in terms of GFLOPS (giga or billions of floating-point operations per second). The auto-tuning process constantly outputs the current and best GLOPS for each configuration as well as the time consumed on a task. At the end of the configuration, the program compiles and evaluates the mean time inference cost of the optimized model.

3.1 WORKFLOW OF TVM



(Diagram 1. High-level diagram of TVM workflow.)

Diagram 1 shows the workflow of the auto-tuning process that starts from a template [12]. In step 1, the program fetches the neural network model, Resnet-50, and translates the model into an intermediate representation (IR) module using its high-level language, Relay. Step 2 is when the program utilizes these modules and respective parameters that have been generated with the module to create tasks on threads within a loop. As the two variables are built, new tasks are being formed. Step 3 shows how the task name and configuration space is formed with *strategy.add_implementation(topi.cuda.conv2d_nchw, topi.cuda.schedule_conv2d_nchw)*; tasks and configuration spaces for NHWC layouts are also similarly initialized. Once the loop from step 2 has completed, it appends the created tasks into a list and returns it to the main body. Each task is then dealt individually by a tuner of choice; here, the XGBoost search algorithm is used. The tuner searches for the most optimal configurations from the initialized configuration space and measures their performance in step 5. Users are able to see specific GFLOPS recordings that on the console. Lastly, the main program compiles and evaluates the inference time cost by referring to the selected configurations in a log file that was generated during tuning. It uses LLVM to translate high-level representations into machine-dependent assembly language and compile to the specific hardware.



(Diagram 2. Space definition of AutoTVM.)

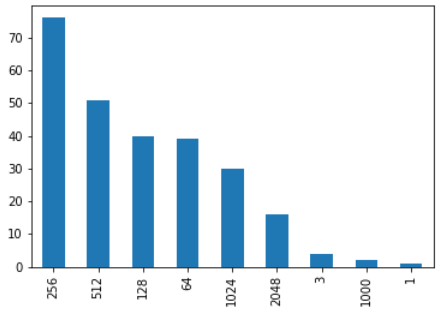
To be more specific on configuration space, Diagram 2 depicts where and how the configuration space is defined; space is defined in either *conv2d_direct.py* or *conv2d_nhwc.py* according to the memory layout. GPUs have a shared memory and programmers can design code to optimally make use of this memory space. The program assigns computation tasks to threads and each thread allocates computations to and from its local memory and shared memory. These threads are bound with a two-dimensional matrix which are split into blocks (sub-matrices) and these blocks are further split into smaller tiles; the splitting is defined via the *schedule_conv2d* methods. The matrices that have been split are computed in nested loops that are proportional to the number of splits. Intuitively, the more tiles there are, the larger the configuration space becomes. If we take NHWC as an example, its configuration space is defined in *schedule_conv2d_nhwc_direct*. Each knob is defined with a specific number of parameters that users can modify. In this case, the configuration space of NHWC is defined with tile *n*, tile *c*, number of thread *n*, and number of thread *c* with three values each. Additionally, it has two parameter values for both virtual thread *n* and *c*. The layout is also assigned with four values for the *step* knob. To calculate the size of the configuration space, we simply multiply the number of values that each knob has. Thus, the configuration space for the NHWC memory layout would be $3 \cdot 3 \cdot 3 \cdot 3 \cdot 2 \cdot 2 \cdot 4 = 1296$. Several values for each knob have been tested and their inference time costs have been compared in this project. The results are explained in detail in the *Results* section of this paper.

The main differences between tuning NCHW and NHWC layouts are their tasks and tensor values. It is important to differentiate between the two memory layouts before attempting to merge their configuration space and understanding their tensor values helps determine what parts of the compiler to change for merging. Tensor values are created for each memory layout

and the figures below describe the general shape of a tensor. For example, a tensor is stored in the form (1, 3, 3, 224).

	0	1	2	3	4
0	data	1	3.0	224.0	224.0
1	bn_data_gamma	3	NaN	NaN	NaN
2	bn_data_beta	3	NaN	NaN	NaN
3	bn_data_moving_mean	3	NaN	NaN	NaN
4	bn_data_moving_var	3	NaN	NaN	NaN
..	...	...	...	...	...
254	bn1_beta	2048	NaN	NaN	NaN
255	bn1_moving_mean	2048	NaN	NaN	NaN
256	bn1_moving_var	2048	NaN	NaN	NaN
257	fc1_weight	1000	2048.0	NaN	NaN
258	fc1_bias	1000	NaN	NaN	NaN

[259 rows x 5 columns]



(Figure 10. Parameter names with tensor values (left) and distribution of tensor values (right))

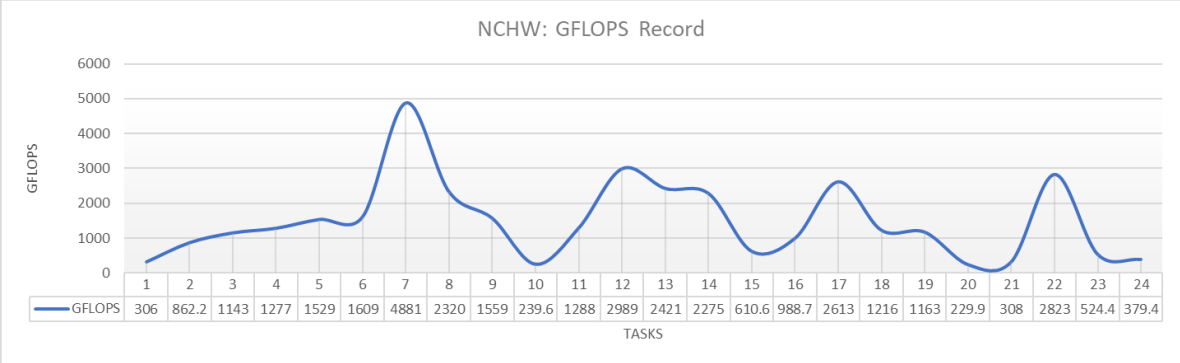
The first column lists the names of parameters that have been generated according to the module, and both NCHW and NHWC layouts have the same parameter names when they are formed in their tuning processes. They differ in the position of the tensors but contain the same values, meaning a simple transposition would make them compatible for merging. It needs to be noted that these tensors are not generated from Tensorflow but are objects of a class called *Tensor* defined in the TVM compiler, and values are stored as tuples. Tuples are immutable after initialization, so the *tvm.topi.transpose* function is required for transposition.

3.2 N-TRIAL AND EARLY STOPPING

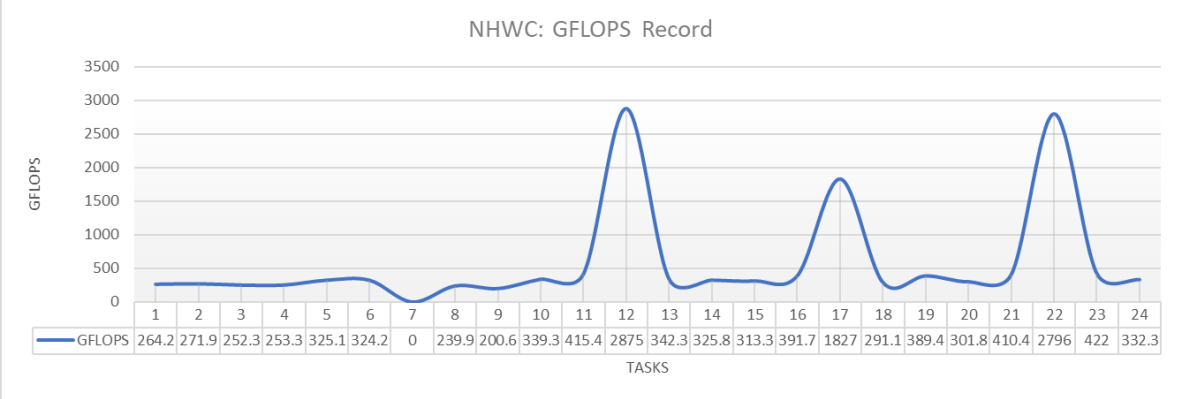
N_trial and *early stopping* are basic parameters that are easily adjusted by assigning integer values to them. The former refers to the total number of parameters that the search algorithm tries while the latter stops the search algorithm if a certain number of trials does not achieve higher GFLOPS (*early stopping* was set to 80% of *n_trials* in this research). The following two graphs in this section measure the GFLOPS performance in NCHW and NHWC layouts, each with *n_trials* set to 500. The last figure shows how using multiple GPUs decreased the tuning time and resulted with less fluctuating GFLOPS.

The x-axis in Figure 9 shown below indicates the task numbers. In each tune, the compiler tunes against the GPU for 24 tasks. Every task has a configuration space that the search algorithm, XGBoost, explores and makes run-time measurements in GFLOPS units.

Tuning the NCHW layout of ResNet-50 took 14.17 hours to complete, which is 2,216.16 seconds in average per task for merely 500 configurations on 24 tasks and three debug phases. The average GFLOPS that each task achieved was 1481.44 GFLOPS. As illustrated in Figure 5 below, GFLOPS fluctuate frequently in different tasks, in which the highest out of all the tasks reached to approximately 4,881 GFLOPS (according to NVIDIA, the TITAN Xp GPU is capable of 12 TFLOPS, or 12,000 GFLOPS [7]).



(Figure 9. Highest GFLOPS per task, NCHW layout, *n_trials* = 500)



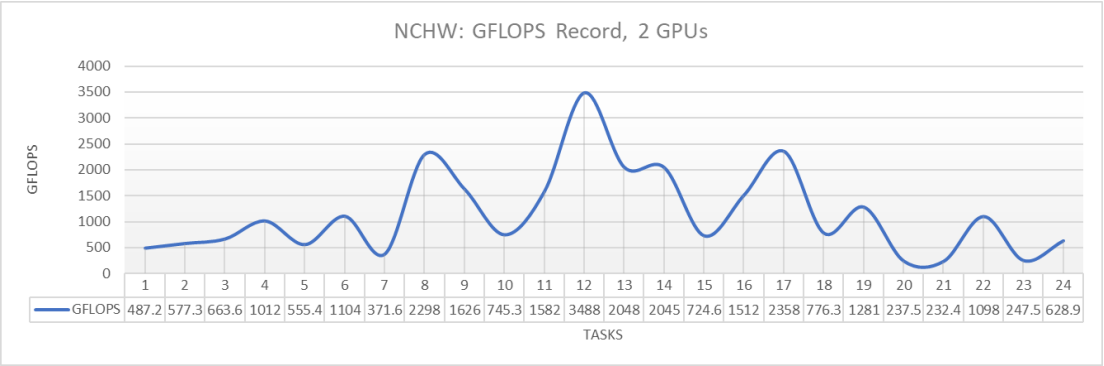
(Figure 10. Highest GFLOPS per task, NHWC layout, *n_trials* = 500)

On the other hand, configuring the ResNet-50 in NHWC layout took less time for the 24 tasks. The highest GLFOPS that could be reached was 2874.88 GFLOPS, which is significantly lower than that of the NCHW configuration. It was not a surprise but rather expected from a NHWC memory layout since graphics cards are more efficient on a NCHW memory layout. The GFLOPS have also fluctuated but, since individual tasks have different objectives and thus differences in performance, it is arguably a natural phenomenon. The NHWC model had an inference time cost of 10.86 milliseconds.

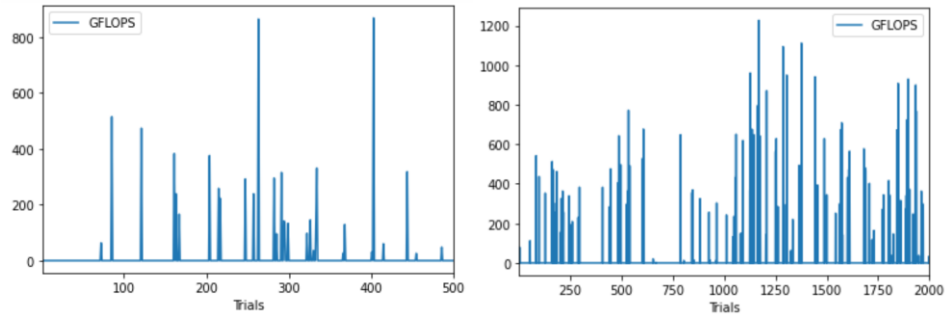
Robust configurations lead to drastically improved performances by exploring the most optimal knob values. Indeed, increasing that number of configuration processes from 500 to 1000 or even 2000 would give the program the opportunity to explore more sets of configurations that is best fit for the GPU. The tuning time can be further reduced by connecting multiple high-end GPUs to the RPC tracker.

Tuning with two GPUs took 12.14 hours of tuning. Although the average GFLOPS per task turned out to be 1154.2 only, the standard deviation of GFLOPS was lower by approximately 300 GFLOPS, meaning that there was less fluctuation. Yet, using multiple GPUs only speeds up the process and does not greatly affect the maximum performance that it can attain. Increasing the number of configuration test per task is the better solution to aim for higher

GFLOPS. A higher number of CPU cores and GPUs assigned to one tuning process could significantly reduce the configuration time as well.



(Figure 11. Auto-tuning the NCHW layout with two GPUs.)



(Figure 12. 500 trials (left) and 2000 trials (right) on NCHW layout)

Figure 12 shows the GFLOPS achieved in each configuration trial on NCHW layout for a single task out of the 24. The highest GFLOPS was 868.3 for 500 trials and 1226.64 for 2000 trials. 1000 trials could also achieve 1145.09 GFLOPS. Each of the 500, 1000, and 2000 trials had an inference cost of 0.35, 0.23, and 0.25 milliseconds. In other words, the most optimal number of configuration trials turned out to be approximately 1000 trials for a specific task under a time constraint.

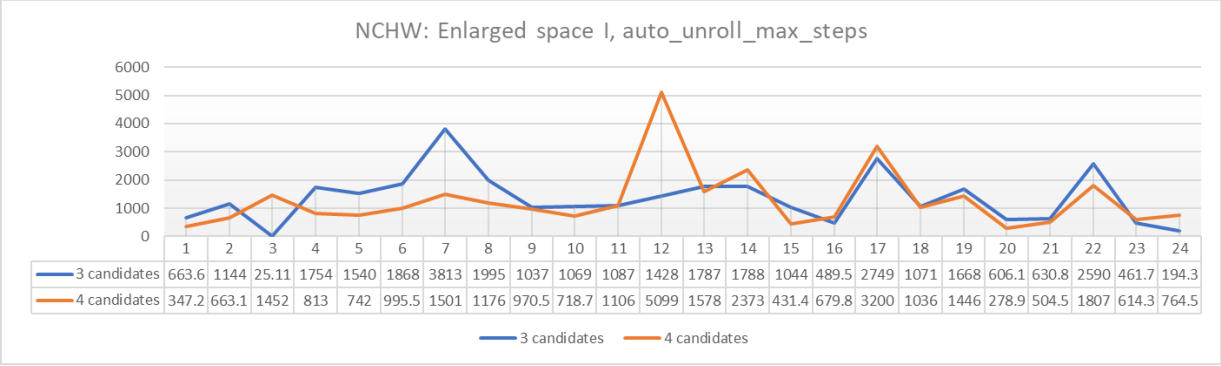
3.3 ENLARGING THE DESIGN SPACE I: INCREMENTATION

AutoTVM is a compiler in which users can define specific parameter candidates in the configuration space and optimally customize their design space. In this section, a variety of parameter combinations has been tested to witness their effects on performance in NCHW and NHWC memory layouts separately and evaluate the current setting of the configuration template that is defined in TVM. Specifically, NCHW's *auto-unroll-max-step* and NHWC's number of tiles and virtual threads have been tested and the results have shown that additional parameter values generally lead to increased computational performance and lower inference time costs.

The experiment was conducted by first running a tuning process with the default configuration space as currently defined in TVM's GitHub repository to set the baseline for comparison, which had two to three candidate values in the *define_knob* functions. Then, an additional candidate value would be added to the knob in the NCHW design space (now the total number of candidate values for a specific *define_knob* function has increased to four) and the GPU is tuned from the beginning again with their GFLOPS performances recorded at the end of each complete tune. A total of seven candidate values were experimented for several knobs in the NHWC design space under the same method. Usually, each complete tune would take approximately 10 hours under 1000 trials, but, for the sake of time, 100 trials was set for each tune.

Although there needs to be additional experimentation on a wider range of *auto_unroll_max_step* numbers for a more insightful view of the parameter's impact on performance, it was determined that the inference cost was significantly reduced with an additional *auto_unroll_max_step*.

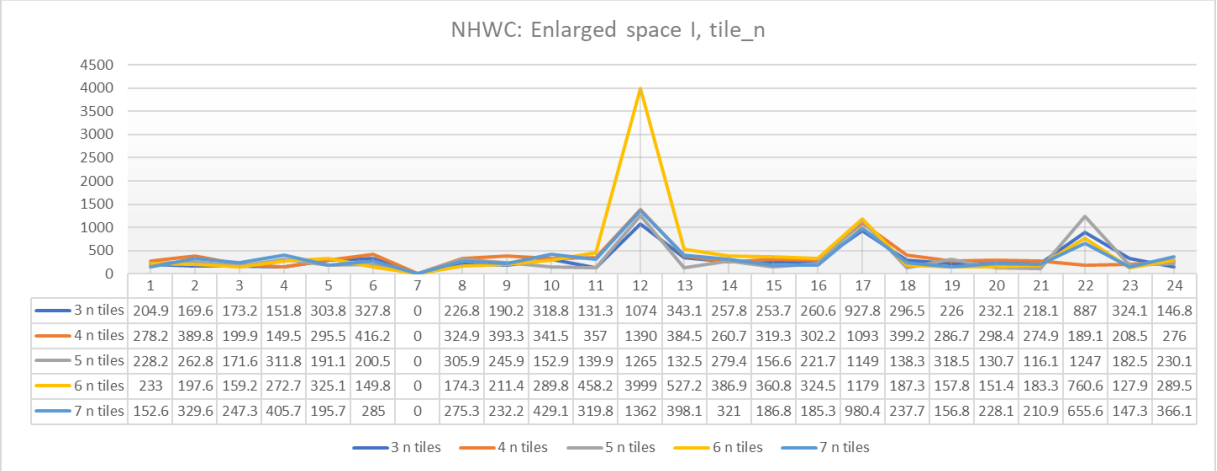
Moreover, increasing the number of *tile_n* of NHWC's design space could achieve higher GFLOPS for tasks 12, 17, and 22. Inference time cost has also decreased the more *tile_n* there were. Extra *tile_c* have also proven to bring higher GFLOPS for the same tasks, but did not largely affect the inference time cost.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3 candidates	663.6	1144	25.11	1754	1540	1868	3813	1995	1037	1069	1087	1428	1787	1788	1044	489.5	2749	1071	1668	606.1	630.8	2590	461.7	194.3
4 candidates	347.2	663.1	1452	813	742	995.5	1501	1176	970.5	718.7	1106	5099	1578	2373	431.4	679.8	3200	1036	1446	278.9	504.5	1807	614.3	764.5

(NCHW: GFLOPS per number of *auto_unroll_max_step*)

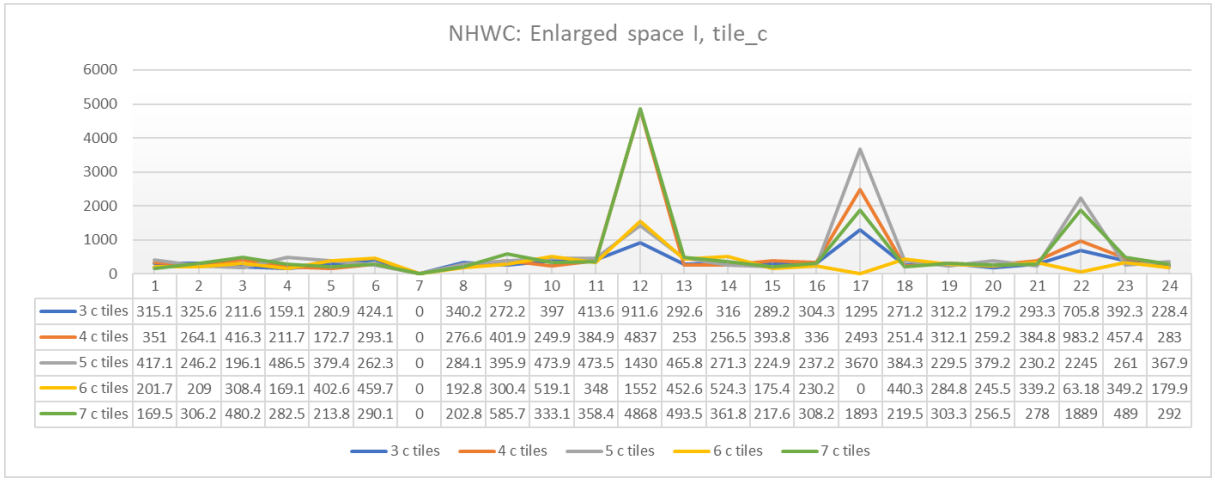
Auto-tuning in the NCHW layout with three and four *auto_unroll_max_step*s showed varying performances. To elaborate on the function of this variable, loop-unrolling is a programming technique that shortens a loop by condensing a loop with pre-calculations and an offset to manage the new number of loop iterations. Condensing the loop is, thus, called unrolling; it would be analogous to unwinding a spring.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3 n tiles	204.9	169.6	173.2	151.8	303.8	327.8	0	226.8	190.2	318.8	131.3	1074	343.1	257.8	253.7	260.6	927.8	296.5	226	232.1	218.1	887	324.1	146.8
4 n tiles	278.2	389.8	199.9	149.5	295.5	416.2	0	324.9	393.3	341.5	357	1390	384.5	260.7	319.3	302.2	1093	399.2	286.7	298.4	274.9	189.1	208.5	276
5 n tiles	228.2	262.8	171.6	311.8	191.1	200.5	0	305.9	245.9	152.9	139.9	1265	132.5	279.4	156.6	221.7	1149	138.3	318.5	130.7	116.1	1247	182.5	230.1
6 n tiles	233	197.6	159.2	272.7	325.1	149.8	0	174.3	211.4	289.8	458.2	3999	527.2	386.9	360.8	324.5	1179	187.3	157.8	151.4	183.3	760.6	127.9	289.5
7 n tiles	152.6	329.6	247.3	405.7	195.7	285	0	275.3	232.2	429.1	319.8	1362	398.1	321	186.8	185.3	980.4	237.7	156.8	228.1	210.9	655.6	147.3	366.1

(NHWC: GFLOPS per number of *n* tiles)

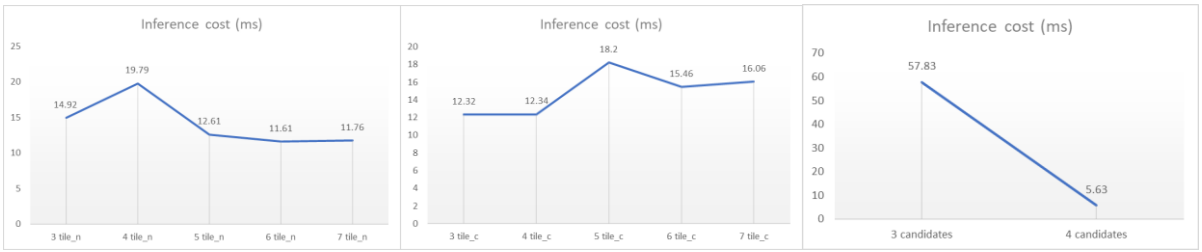
Increasing the number of *tile_n* candidates showed slightly higher GFLOPS. Taking Task 12 as an example, it is seen that three candidates of *tile_n* were measured with 1,074 GFLOPS. Increasing the number of candidates to four in the next tune resulted with 1,390 GFLOPS, five candidates resulted with 1265 GFLOPS, 6 candidates resulted with 3,999 GFLOPS, and seven candidates resulted with 1,362 GFLOPS. There are similar trends shown in tasks 17 and 22 as well.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3 c tiles	315.1	325.6	211.6	159.1	280.9	424.1	0	340.2	272.2	397	413.6	911.6	292.6	316	289.2	304.3	1295	271.2	312.2	179.2	293.3	705.8	392.3	228.4
4 c tiles	351	264.1	416.3	211.7	172.7	293.1	0	276.6	401.9	249.9	384.9	4837	253	256.5	393.8	336	2493	251.4	312.1	259.2	384.8	983.2	457.4	283
5 c tiles	417.1	246.2	196.1	486.5	379.4	262.3	0	284.1	395.9	473.9	473.5	1430	465.8	271.3	224.9	237.2	3670	384.3	229.5	379.2	230.2	2245	261	367.9
6 c tiles	201.7	209	308.4	169.1	402.6	459.7	0	192.8	300.4	519.1	348	1552	452.6	524.3	175.4	230.2	0	440.3	284.8	245.5	339.2	63.18	349.2	179.9
7 c tiles	169.5	306.2	480.2	282.5	213.8	290.1	0	202.8	585.7	333.1	358.4	4868	493.5	361.8	217.6	308.2	1893	219.5	303.3	256.5	278	1889	489	292

(NHWC: GFLOPS per number of *c* tiles)

The affect of increasing the number of candidates is clearer with the *tile_c* variable. Four and seven candidates of *tile_c* could measure 4,837 and 4,868 GFLOPS while three candidates could only measure 911 GFLOPS.



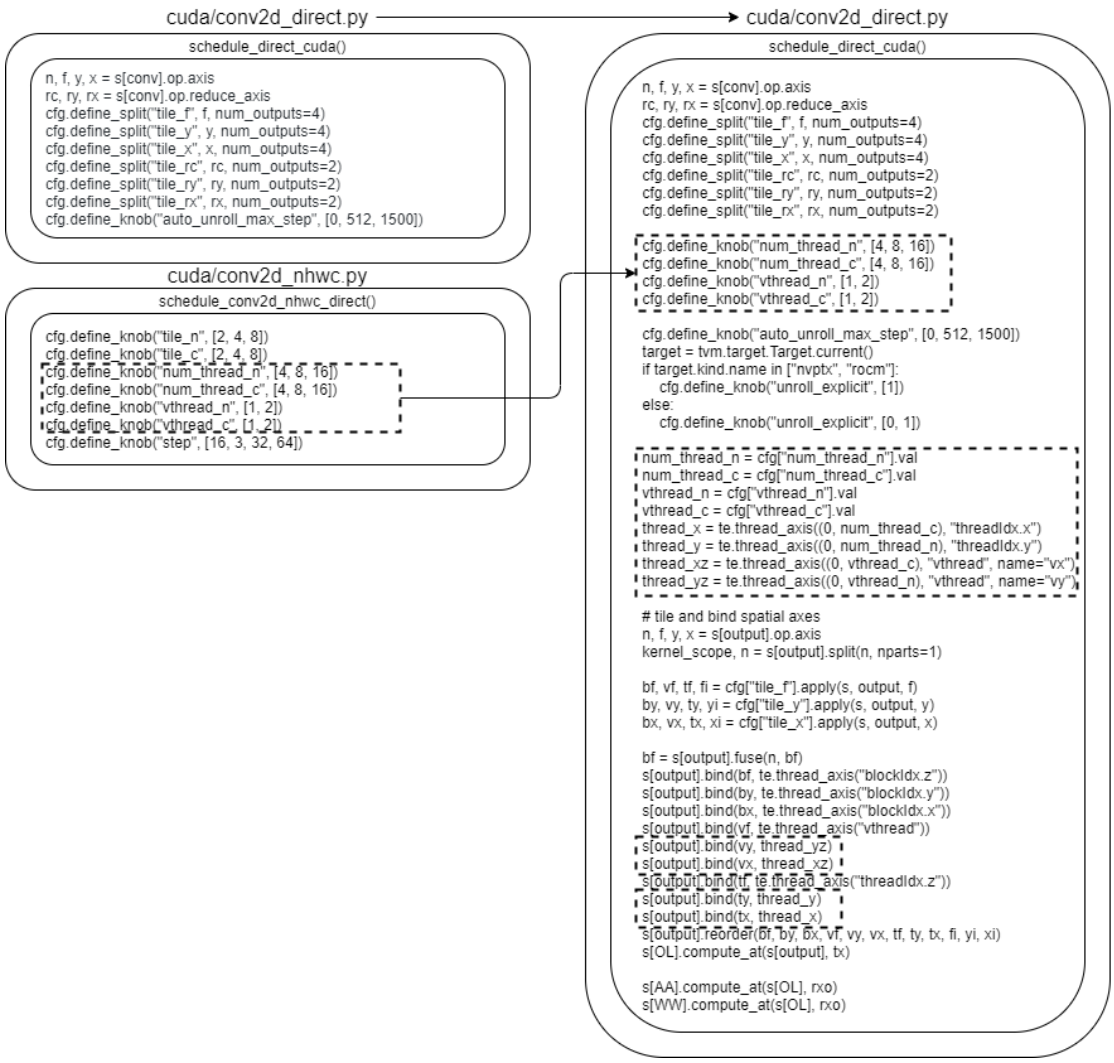
(Inference time cost of NHWC *n* tiles (left), *c* tiles (middle), and NCHW *auto unroll* (right))

After enlarging the design space by incrementing the number of candidate values, there were noticeable performance boosts in GFLOPS. There needs to be more investigation into why task 7 recorded 0 GLFOPS, however. Inference costs also decreased for both *tile_n* and *auto_unroll* variables while *tile_c* did not show lower inference time costs.

3.4 ENLARGING THE DESIGN SPACE II: INTRODUCTION OF NEW KNOBS

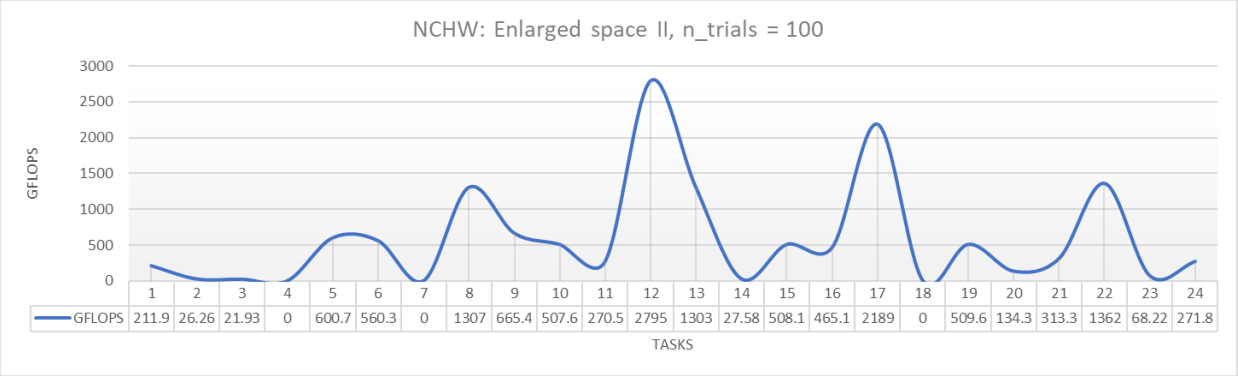
To reiterate, AutoTVM is a template-based compiler, meaning each memory layout has its individual space definition. For instance, NCHW has a design space that is defined with splitting tiles *f*, *y*, *x*, *rc*, *ry*, and *rx*, while the space of NHWC is defined with knobs tile *n*, *c*, number of threads, virtual threads, and *steps*, as shown in Diagram 2 on page 8. Hence, the default template only includes a small set of parameters for the memory layouts, limiting the explorable search space of the search-algorithm unless the number of candidate values is incremented. Thus, the design space of NCHW has been enlarged by implementing NHWC configuration knobs such that the search-algorithm is able to compute a wider range of configurations. Consequently, the compiler was able to attain higher GFLOPS for ResNet-50.

Knobs *auto_unroll_max_step* and *unroll_explicit* of NCHW could not be integrated into the NHWC design space because they require a specific kernel scope, which is currently not provided by the default configuration space template.



(Diagram 3. Insertion of NHWC tunable knobs into the NCHW configuration template)

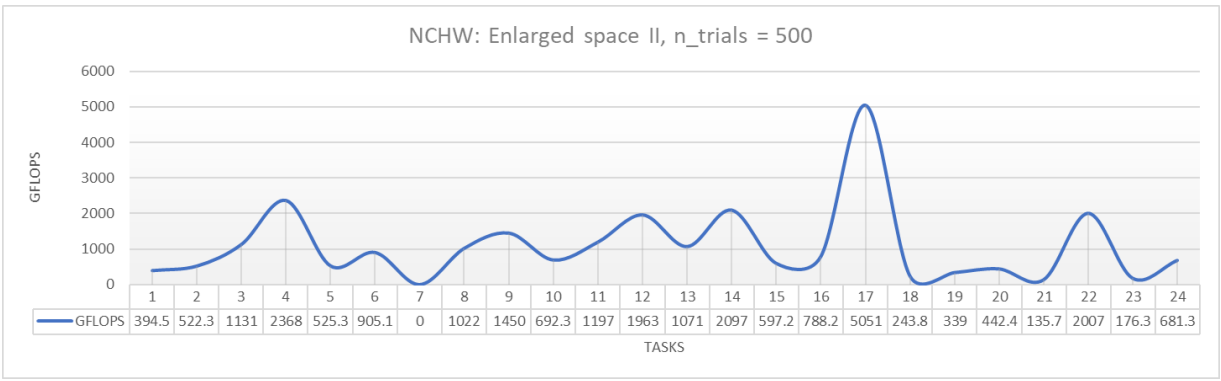
The enlarged design space then had a total of 12 variables to explore after adding *num_thread_n*, *num_thread_c*, *vthread_n*, *and vthread_c*. The design space would then have tiles *f*, *y*, *x*, *rc*, *ry*, *rx*, *auto_unroll_max_step*, *unroll_explicit*, number of threads, and virtual threads as shown in a screenshot later. This has increased the configuration length from a couple million to billions. A higher peak GFLOPS performance was recorded, comparing Figure 14 to Figure 9. The number of candidate values were not modified in this section.



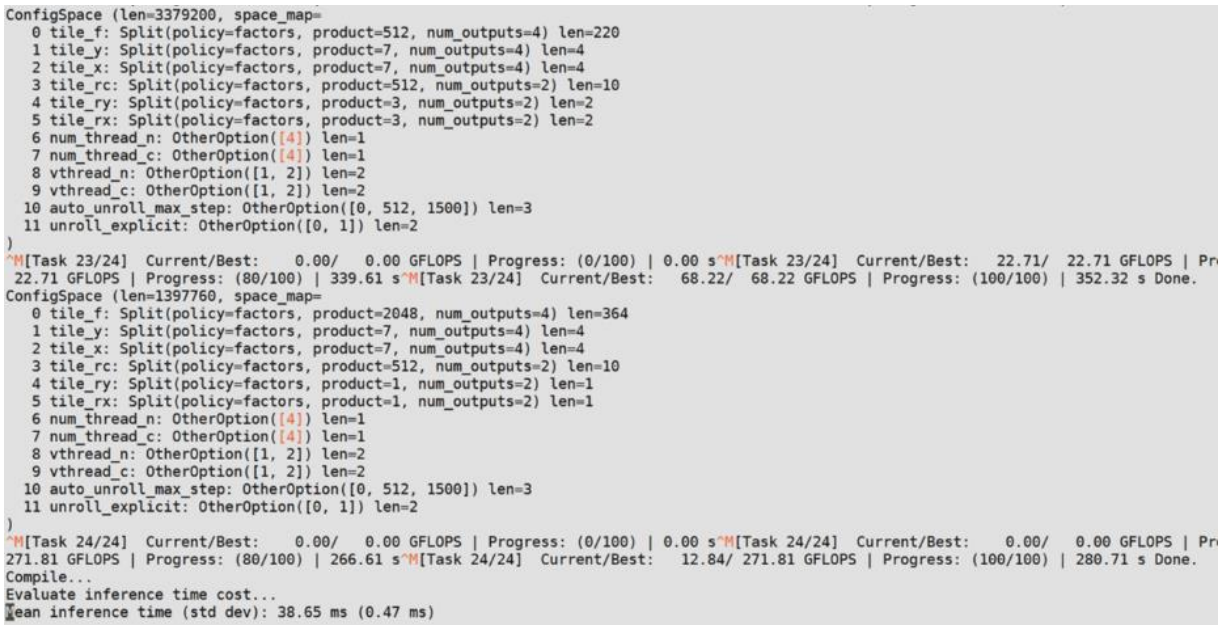
(Figure 13. GFLOPS record of an enlarged space II, where *n_trials* = 100.)

Figure 13 shows the GFLOPS for each task with *n_trials* set to 100. It was noted that tasks 4, 7, and 18 have recorded 0.00 GFLOPS each, but tasks 4 and 18 recorded non-zero GFLOPS as the value of *n_trials* was increased to 500 as seen in Figure 14. Task 7, on the other hand, did not indicate higher GFLOPS even after increasing the number of trials to 500. Nevertheless, since this case is also seen in Figure 10, the problem is not related to the introduction of new knobs into the design space.

N_trials was set to 500 for the second tune and 5050.64 GFLOPS was recorded at task 17 under the design space with extra knobs. Additional tuning is yet to be made for a more accurate comparison of the performance differences of design spaces but providing a wider range of configurations allowed for a more comprehensive exploration, nevertheless. The inference costs of tuning at *n_trials* = 100 and *n_trials* = 500 are 10.86 milliseconds and 7.41 milliseconds, respectively.



(Figure 14. GFLOPS measures of an enlarged space II, where *n_trials* = 500.)



(List of configuration variables and final inference cost of a diversified space.)

The picture above is a screenshot of tuning the enlarged configuration space and the new knobs "*num_thread_c*", "*num_thread_n*", "*vthread_n*", and "*vthread_c*" can be seen. It needs to be noted that fallback configurations cannot be used in the case of tuning diversified configuration spaces because fallback configurations are fetched during runtime from the TVM GitHub repository at *https://raw.githubusercontent.com/tlc-pack/tophub/main/tophub/cuda_v0.09.log* and NHWC knobs are not present in the list of NCHW fallback configurations. Thus, fetching fallback configurations needs to be disabled if diversified design spaces are to be explored unless the repository itself is updated with the new parameters of interest. Yet, the introduction of new knobs into the NCHW design space has enlarged the configuration length and was able to attain a higher peak GFLOPS when compared to tuning the NCHW default design space with the same number of *n_trials*.

4 ATTEMPTS ON MERGING DESIGN SPACES

The current version of the TVM compiler tunes models of different memory layouts individually. This limits the variety of configurations that the program can explore and broadening the configuration scope would facilitate search-algorithms to identify more optimal configurations for the hardware. Several attempts have been made to achieve this objective and they have been briefly recounted below.

TVM first loads a neural network model and converts it into an intermediate representation (IR) with Relay. The representation is stored and passed around in the program in the form of a Relay IR module and the compiler extracts attributes of the module, such as memory layout and kernel shape. It then extracts tasks according to these attributes; if the attribute has a memory layout of NCHW, it will bind a NCHW task name (*conv2d_nchw.cuda* or

conv2d_nchw_winograd.cuda) and NCHW schedule as part of the tuning strategy. The attempt was to bind NHWC task names and schedules to the NCHW-layout module during strategy implementation, increasing the number of tasks from 24 to 48. However, the newly added tasks were skipped when tuning due to GPU kernel mismatch between the NCHW module and NHWC tensor values or schedules.

Another attempt was reading the log file that was generated from sequentially tuning two types of modules with different layouts in a single run and unifying the tensor formats within the log file. More specifically, the log file stores the best configurations on each line in dictionary format. The values that differed between the two memory layouts were the task name, tensor values, and configuration entities, and a code was written that would unify their task name and tensor values by replacing and transforming or reversing tensor values. But it was noticed that, during the compilation of this newly formatted log file, configuration entities (knobs) of the other memory layout's design space could not be recognized.

5 DISCUSSION

Using the TVM deep learning compiler, the Resnet-50 neural network models in NCHW and NHWC layout were optimized via the enlargement of their configuration spaces. The performances of the models were improved by not only increasing the number of parameter candidates but also introducing new knobs. The XGBoost search-algorithm was able to explore a wider range of configurations in both cases.

In detail, the tuning programs ran for approximately 20 hours. Both NCHW and NHWC memory layouts were configured under various settings and the results showed that the GPU could reach the highest GFLOPS of 4,881.24 GFLOPS with an inference cost of 5.07 milliseconds on the NCHW memory layout under the default design space. On the other hand, the GPU could reach higher at 5,051 GFLOPS on an enlarged configuration space of NCHW with an inference cost of 7.41 milliseconds and incrementing the number of candidate values indicated a general increase in performance for *tile_n* and *auto_unroll_max_step* variables for both layouts.

Several attempts have been made to merge the design spaces of NCHW and NHWC but there was no success. Nevertheless, the insight gained through these attempts will still be valuable to future research on merging design spaces of different memory layouts. This research paper may also serve as reference to the performance differences between increasing the number of candidates and introducing new knobs into a design space. In addition, auto-scheduler: Ansor is a new component of the TVM compiler that allows tensor programs to be generated on-the-fly [11]. This is a major advantage since it eliminates the need for users to manually define a limited configuration space. Ansor is capable of creating and exploring a much comprehensive design space via hierarchical representation and evolutionary search with shorter tuning time due to efficient algorithms [11]. Therefore, Ansor is a sensible alternative to AutoTVM.

6 REFERENCES

1. Chen, T., Moreau, T., Jiang, Z., Zheng, L., & Yan, E. (2018). TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation* (pp. 579-594). Carlsbad: USENIX Association. Retrieved from https://www.usenix.org/system/files/osdi18-chen.pdf
2. Common architectures in convolutional neural networks. (2020). Retrieved 26 November 2020, from https://www.jeremyjordan.me/convnet-architectures/#resnet
3. Difference between NCHW and NHWC in TensorFlow API - Programmer Sought. (2020). Retrieved 23 November 2020, from https://www.programmersought.com/article/77463626294/
4. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *Microsoft*, 1-12. Retrieved from https://arxiv.org/pdf/1512.03385.pdf
5. Li, M., Liu, Y., Liu, X., Sun, Q., You, X., & Yang, H. et al. (2020). The Deep Learning Compiler: A Comprehensive Survey. *DeepAI*, *1*(1), 1-34.
6. Ng, A. (2020). China tightens control with facial recognition, public shaming. Retrieved 29 November 2020, from https://www.cnet.com/news/in-china-facial-recognition-public-shaming-and-control-go-hand-in-hand/
7. NVIDIA TITAN Xp Graphics Card with Pascal Architecture. (2020). Retrieved 25 November 2020, from https://www.nvidia.com/en-us/titan/titan-xp/
8. ResNet-50 convolutional neural network - MATLAB resnet50. (2020). Retrieved 24 November 2020, from https://www.mathworks.com/help/deeplearning/ref/resnet50.html
9. Xing, Y., Weng, J., Wang, Y., Sui, L., Shan, Y., & Wang, Y. (2019). An In-depth Comparison of Compilers for Deep Neural Networks on Hardware. *IEEE*. doi: 978-1-7281-2437-7
10. Zahran, M. (2017). Heterogeneous Computing: Here to Stay. *ACMqueue*, *14*(6), 1-12. Retrieved from https://queue.acm.org/detail.cfm?id=3038873
11. Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, H., Hal-Ali, A., . . . Stoica, I. (2020). *Ansor: Generating High-Performance Tensor Programs for Deep Learning*. OSDI.
12. Zheng, L., & Yan, E. (n.d.). Auto-tuning a convolutional network for NVIDIA GPU. Retrieved September 18, 2020, from https://tvm.apache.org/docs/tutorials/autotvm/tune_relay_cuda.html