Design of FPGA-Based Deep Learning Accelerator with TVM Compiler

Submitted To

Dr. Michael Orshansky

Prepared By

Elgin Allen
Sergio Chacon
Troy Jackson
Jeffrey Marshall
Michael Pontikes
Zachary Sisti

EE464 Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Fall 2020

CONTENTS

CONTENTS (Continued)

TABLES

FIGURES

EXECUTIVE SUMMARY

The purpose of this report is to cover the results found from testing the efficacy of the Tensor Virtual Machine (TVM) compiler stack. The TVM compiler is a Deep Learning (DL) compiler that focuses on flexibility, specifically being capable of taking multiple frontend frameworks, compiling those, and then deploying inference applications to hardware backends. As a part of the stack, the Versatile Tensor Accelerator (VTA) dual-ISA architecture is used to program the Field Programmable Gate Array (FPGA). In the case of this report, the full capabilities of the TVM compiler, with the inclusion of the VTA portion of the software and hardware stack, are tested. The project focused on research of the compiler's performance in terms of inference times. The compiler leverages tuners and schedulers to improve performance for different hardware backends.

To test the capabilities of the TVM compiler, we used a top down structure, beginning with the design of a system based on the compiler, then extending that system to an FPGA, adding in the VTA architecture, deciding on a specific Convolutional Neural Network (CNN) to test, and then finally evaluating the tuners and schedulers provided by the compiler. We combined each of these pieces into a full system that we could research to see different performance results based on different characteristics provided by the compiler. In implementing our design, we decided to use the CNN ResNet so we could see performance changes with different sized CNNs, since ResNet comes in different sized versions. Implementing the necessary steps to connect these components was a lengthy task that involved a main Python control script, a VTA bitstream, and a Remote Procedure Call (RPC) server.

In implementing the system, we encountered several issues that reshaped our approach to the project. For example, upon learning that the x86 laptop CPU was not an accurate point of comparison to the VTA hardware, we decided to restart our test and evaluation process to perform those tests on the PYNQ board's SoC CPU instead. Since the TVM website used the Xilinx PYNQ board in its examples and tutorials, we selected this as the FPGA for our implementation. This put restrictions on our work, as connecting to the PYNQ board required very specific circumstances. We also encountered many roadblocks resulting from issues with the TVM compiler itself, since it is still in development. Most notably, an AutoTVM bug halted our progress, forcing us to locate the faulty code and resolve it ourselves.

We evaluated the performance of each layer of our system and found that the use of an FPGA programmed with the VTA architecture greatly improves inference time, while using a CPU on its own is significantly slower. In the case of ResNet-18, VTA is 2x better than the CPU, while ResNet-50 sees a 5x improvement. We also found that the different autotuner configurations did not significantly improve inference time when compared to the default tuned configuration. After testing the tuners we created our own custom bitstreams, only to find that different clock speeds did not lead to any significant change in performance. Additionally, after examining the resource allocation within the PYNQ board, we discovered that we had an exceedingly high Look-up Table (LUT) usage and an exceedingly low Block Random Access Memory (BRAM) usage.

During our project we stayed well within our budget constraints. Our main cost consisted of the PYNQ board we ran inference on, costing about $150. We were later presented with another PYNQ board by Dr. McDermott, which was a considerable resource. We did, however, run into

two significant time sinks. The first consisted of multiple issues with TVM's RPC server, which we eventually solved by both consulting the TVM forums and identifying bugs independently. The second was running AutoTVM's tuning algorithms, which took many hours for each datapoint, and had to be restarted upon any significant network fluctuation.

Our major safety and ethical concerns when approaching our project were those surrounding bias in machine learning. This bias is created, often involuntary, when only select data groups are used to train the model. While our models were mainly picked from areas that do not have a perceived bias, we understand that this is a large issue in the machine learning area and is an issue that needs to be addressed. In our research, we strive to avoid human bias as much as possible.

For further research, we would recommend teams look into the VTA architecture itself, as it has major memory bottlenecks that could be improved upon by leveraging the BRAM in the FPGA and reducing the high LUT utilization. The autotuner for TVM, AutoTVM, could also be researched further to see how performance changes with different CNNs. Overall, our results showed that leveraging an FPGA for inference applications leads to great performance increases, despite VTA's suboptimal resource usage. Consequently, the architecture should be improved or recreated in a better form.

1.0 INTRODUCTION

This document provides an overview of our project by examining our previous work, and drawing conclusions from our results. The following sections comprise a technical description of our design problem, an explanation of our solution, and the details of our implementation and testing. The report closes with our comprehensive considerations and recommendations for future improvement.

With increasing research into deep learning, there has also been an increase in different frameworks and hardware backends. This has led to the need for a flexible compiler that can take those frameworks as an input and deploy modules from those frameworks to diverse hardware backends. The Tensor Virtual Machine (TVM) compiler and Versatile Tensor Accelerator (VTA) architecture were designed to fill this need.

In this project, we were tasked with testing the efficacy of TVM and VTA to run a Convolutional Neural Network (CNN) for image classification on the PYNQ Field Programmable Gate Array (FPGA) board. To accomplish this, we split into software and hardware sub-teams and began by implementing TVM and VTA. We then researched CNN models and selected ResNet to benchmark the system, after which we developed, refined, and executed a test plan. We compared inference time with and without VTA and found that it significantly accelerates the process. Additionally, we tested different tuning methods and clock speeds and generated synthesis reports to locate potential areas for improving the VTA architecture. Although we were ultimately unable to perform these optimizations due to time constraints, our findings pave the way for further performance improvements in image classification with the TVM stack.

2.0 DESIGN PROBLEM

This section describes the context and purpose of our project, providing background information on key subjects and explaining the significance of our research. In this section, we also define our design problem and the specific requirements met by the team in completing our project.

2.1 Background Information

A preliminary examination of our project's technical context is necessary to understand its purpose and the approach we took in completing it. In the following subsections, we provide background on the fields of Deep Learning (DL) and hardware acceleration, as well as an explanation of their current importance. This background allows for a discussion of these concepts' relevance to the project.

2.1.1 Deep Learning

Machine Learning (ML) is the process of a computer learning a task by analyzing data sets and training models. DL is a subset of ML in which the computer constructs a neural network that changes throughout its processing of data sets [1]. The computer maintains and updates this dense network of nodes as it processes data by assigning weight from incoming nodes on the directed network flow. These neural networks can take on a variety of sizes and shapes, as shown in Figure 1, below, where the deep learning neural network contains more layers and more nodes, achieving a more developed and precise output. The computer multiplies the value of the data coming through a node by the associated weight to receive a number. This number is used to determine whether or not to fire the current node and send the data to all connected nodes. Over time, this yields a neural network consisting of all the data values passed.

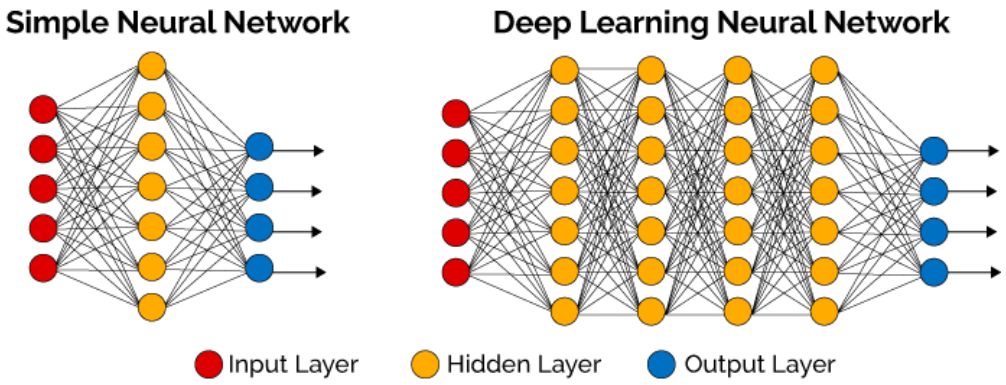

Figure 1. Basic Networks of ML and DL [2]

In this project we deployed a CNN, which is a class of deep neural network generally used to analyze visual inputs. Our project required us to select and implement a CNN for image classification, the most common application of CNNs.

2.1.2 FPGA Hardware Acceleration

In recent years the capabilities of CNNs have significantly increased, and as a result many models suffer from high computational and storage complexities. Currently, Graphical Processing Units (GPUs) are the leading computing platform choice as they offer peak performance in terms of operations per second. However, FPGAs are seen as a promising alternative platform.

FPGA devices use programmable interconnects to bridge matrices of configurable logic blocks [3]. This allows designers to adjust a large majority of the devices' electrical functionality, which includes changing the functionality on every power-up of the device. This ability makes FPGAs some of the most flexible integrated circuits in the semiconductor industry. The flexibility of these devices has led to their use for CNN acceleration. Some of the other benefits of using FPGAs for DL applications are their energy efficiency and ability to increase parallelism [4].

Our project required us to benchmark TVM, a new open-source compiler from the University of Washington that optimizes and schedules neural network models for a variety of hardware backends. A significant portion of this research involved VTA, a DL accelerator template designed specifically to extend TVM to FPGAs.

2.2 Project Goals and Requirements

Our project's goal was to test the performance of the TVM compiler and VTA architecture by implementing a CNN for image classification on an FPGA. Ultimately, this meant our work was oriented significantly more towards research than design. However, we did have requirements to meet in our execution of the project. We needed to select a compatible CNN, successfully compile it with TVM, and run it on the synthesized VTA architecture. We also needed to develop tests to determine the strengths and weaknesses of TVM and VTA. These tests required us to perform many image classification trials and measure the time the CNN took to perform inference.

Due to the research-oriented nature of our project, there were no specific quantitative specifications for us to meet. In fact, our goals evolved as we gained a deeper understanding of

the system, its capabilities, and its limitations. For example, when we discovered that increasing clock speed had negligible impact on the inference time, we focused more attention on tuning methods and locating bottlenecks in the architecture itself.

3.0 DESIGN SOLUTION

The following section provides a detailed outline of our design. First we walk through the physical steps necessary to implement it. Then, we examine each module, breaking down the functionality and overall importance of each. The walkthrough refers to the subsystems by explaining what steps need to be completed for each to create a working accelerator. The module descriptions on the other hand, go into great detail to describe the concepts behind each subsystem and their role in the accelerator.

3.1 Walkthrough

To begin the project, we installed TVM and all necessary dependencies. Then, we created a Python script to handle importing, compiling, and deploying the CNN. Our pre-trained CNN, ResNet50, was imported to our Python script from MxNet's Gluon Model Zoo. This model is a high level computational graph defining the CNN. Our script then transformed this representation into a Relay graph. Relay is simply another library for defining computational graphs, and we used it because of its compatibility with TVM. During this transformation, we applied an AutoTVM log file to specify the scheduling for hardware and graph operators. Next, we applied an eight-bit quantization for the purpose of compression and acceleration. Then, we performed graph packing and constant folding to prepare the data for tensorization and reduce the number of operators. Finally, we built an object file using Relay, which was ready for deployment onto the board. However, before we could do so, we needed to prepare VTA and synthesize it onto the board's FPGA.

Before we got to using VTA in our Python script, we needed to write a configuration file. Named vta_config.json, this file describes the hardware's variables like version, how to find the device, and its data types. Within our Python script, we specified our device as VTA. Next, we used the VTA library and the aforementioned config file to write a VTA bitstream to the board, instructing

it to set up the VTA architecture. Our script was now almost finished, the only thing left to do was create a remote connection from which the script could communicate with the board.

In a different terminal, we used a Secure Shell (SSH) to connect to the board itself and started a Remote Procedure Call (RPC) server. Back on the host terminal script, we specified the IP and port of the RPC server started on the board. Finally, we ran our script to create a connection to the board RPC server so VTA could be synthesized, the Relay object file could be deployed, and from it a graph runtime could be created and executed. Inference results were then sent back to the host terminal for collection.

3.2 Modules

As previously mentioned, our design compiles an image classifying CNN with TVM, and runs it on an FPGA with a synthesized VTA architecture. These subsystems are portrayed in Figure 2 below. The TVM compiler then optimizes the CNN to function efficiently on specific hardware and compiles code into a deployable module. VTA then specifies the architecture for implementation on the FPGA. The final subsystem, the PYNQ board, configures hardware components according to those VTA specifications, builds the graph runtime and runs inference.

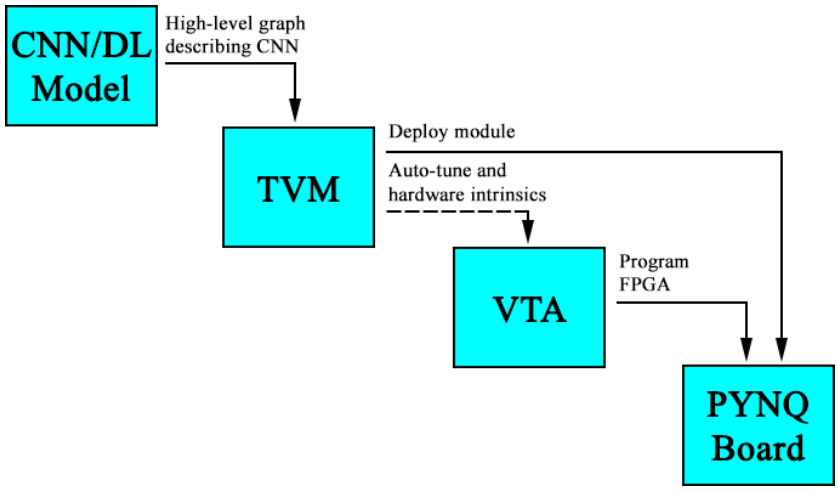

Figure 2. System-Level Overview

3.2.1 CNN/DL Model

The CNN is the central input for our accelerator and the first module we will examine. A CNN is a type of neural network used for image classification and processing. The bulk of the network is made of convolutional layers where input images, represented as matrices, are repeatedly convolved with other matrices to compute the probability of a certain aspect of the image. The results are then fed forward in the network to determine more prominent parts of the image until the image is classified at the end. For example, in a CNN used to detect handwritten digits, early convolutional layers may reveal where on the image a slight curve or sharp edges appear, while later layers may be trained to detect loops of the digit "8" or the curves of the digit "5". The CNN's examined in our project are ResNet-18 and ResNet-50 from MxNet. ResNet networks are successive, in that each builds off of the last. For instance, ResNet-34 includes ResNet-18 and Resnet-50 includes ResNet-34 [5].

3.2.2 TVM

TVM is a DL compiler created to achieve a high level of portability of workloads to a diverse set of hardware backends [6]. The general flow of TVM's usage is to take a CNN, optimize it, and compile it. First, the user imports a high-level computational graph of a CNN from one of many frameworks. Then the user performs graph and code optimizations, to simplify the high-level operations that define the CNN. Operators are then lowered to TVM's TOPI library by creating a schedule and compute function for each. Finally, TVM compiles the schedule into deployable code suitable for the planned hardware.

TVM is the central module of our system. It takes in system inputs and outputs a deployable workload for the FPGA board to later execute. To extend TVM to VTA, the CNN model must be quantized, changed from a float32 data type to int8, and graph packed to correctly format data for the VTA architecture. This process is simple because VTA was created for the express purpose of testing TVM's use with an FPGA board.

3.2.3 VTA

VTA is a customizable deep learning architecture that extends the TVM compiler stack in order to run on FPGAs. Figure 3 on the next page, displays the makeup of the VTA architecture and

consequently what will be synthesized onto the FPGA. VTA makes use of a Task Level Pipeline Parallelism structure to allow for more than one instruction to be fetched or executed at a time [7]. The main methods of communication for the VTA architecture are the queues and buffers and Direct memory accesses (DMA), between modules and from Dynamic Random-Access Memory (DRAM) respectively. The computations either occur on the Tensor Arithmetic-Logic Unit (ALU) or the General Matrix Multiplication (GEMM) core depending on the operation.

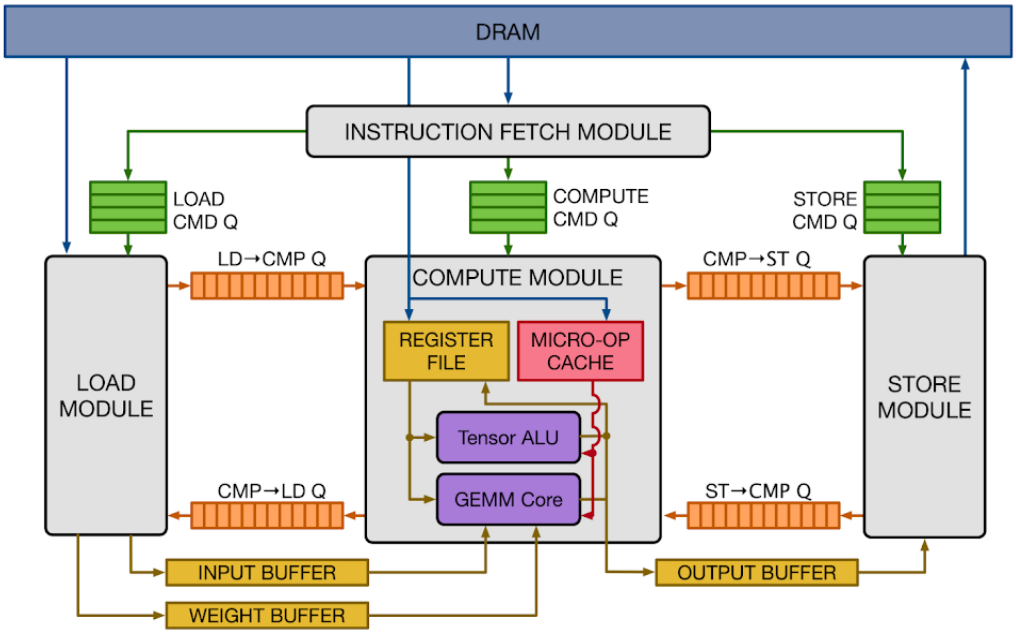

Figure 3. VTA Architecture [7]

The source files for VTA are specified in C++, converted into Verilog files using Vivado High-Level Synthesis, and then synthesized for FPGA implementation. Consequently, each module and piece of hardware in the diagram above corresponds to a written function in the C++ source code. VTA is language bound to Python with which we wrote all of our code. In order to create the FPGA-based deep learning accelerator, operations that occur in the neural network, such as 2D convolutions or simple summations, must be correctly mapped to the VTA architecture in order to improve performance through proper scheduling and resource management.

VTA is a customizable architecture because the parameters and size of the buffers, queues, and compute module can be changed by the designer, thereby allowing for unique implementations. Because of this, working with VTA entails an understanding of how the neural network should operate, an understanding of the hardware modules we implement through software, and the ability to code them in Python.

3.2.4 PYNQ Board

The PYNQ board is a Xilinx device with a Zynq-7000 series SoC (System on a Chip) that can act as an embedded processor through its dual-core ARM Cortex-A9 Processors and as an FPGA through its available Artix-7 family programmable logic. Since an FPGA is programmable, FPGA designs are much more flexible and quicker to implement when compared to a GPU or an Application Specific Integrated Circuit (ASIC). The PYNQ board has the ability to set up a server for remote connection and can be programmed through a MicroSD card to support different hardware and software libraries. For the purposes of this document we will be referring to the PYNQ board's dual-core ARM Cortex-A9 Processor simply as the "SoC CPU".

4.0 DESIGN IMPLEMENTATION

Our project, as described by our faculty mentor Dr. Orshansky always had the TVM compiler and VTA architecture as the center of our design. However, he left us with the choice of which CNN model to benchmark the compiler with and what specific FPGA to run the CNN model on.

The team chose ResNet due to its high inference accuracy along with its less dense architecture when compared to other CNN models. The smaller number of parameters results in less resource constraint issues when being run on our FPGA. However, the model is flexible enough that we are still able to see performance improvements from TVM's optimizations. These improvements are magnified as we test larger versions of ResNet such as ResNet-50 which has several more convolutional layers that can later be optimized and tuned by TVM. There are many FPGAs we could have implemented our design with; however, the PYNQ board is frequently mentioned on the TVM website, and many of their starter tutorials were written for the PYNQ board. The board was well within our $500 budget and also available through Dr. McDermott. For these

reasons, we determined that the PYNQ board would be the simplest and most effective FPGA to utilize as our hardware backend.

As our project developed, our team had to overcome obstacles in our implementation and make adjustments to our preliminary testing. One of the first difficulties we encountered was establishing the RPC connection to communicate remotely with our two PYNQ boards. We had to learn how to set up port forwarding on the PYNQ board's Xilinx server in order to give each member of our team an accessible connection. We also had to bypass the restricting internet firewalls at each of our teammates' apartment complexes. Later on, we had trouble sustaining this remote RPC connection during our initial testing of AutoTVM. After much debugging and continued discussion, we discovered that there was a bug in the TVM library, specifically in one of the written functions for AutoTVM. Essentially, a hidden function was unnecessarily requesting a connection to the already busy PYNQ board, causing a loop as the function waited for a response and eventually creating a time out. This issue had in fact been encountered by other TVM users, but had not been officially resolved until our team found the faulty lines of code. The RPC connection bug was the largest of several inconsistencies found in TVM's documentation and code library, owing to the fact that the compiler is still in development and was regularly updated throughout our project timeline. As a result our team spent a lot of time conversing and posting on TVM's online discussion board to stay up to date with features added to the compiler and methods that other TVM users were using to solve their respective issues.

Two major design alterations our team implemented were moving away from MobileNetv2 as an additional CNN to benchmark and shifting our testing towards the PYNQ board's SoC CPU. Initially, MobileNetv2 was a promising CNN to test due to some of the same reasons we described earlier as motivation to test ResNet. However, as we progressed with our implementation of MobileNet with VTA, we realized that TVM does not currently have sufficient quantization support for MobileNetv2. Without the ability to convert the CNN model from *float32* operations to *int8* operations, inference time would remain high and not be a true representation of the compiler, as the fully leveraged version of VTA is meant to work with a quantized model. After a team discussion, we felt that shifting our focus towards different variations of ResNet and enhancing AutoTVM and VTA would be more worthwhile than the

large time cost of writing our own quantization support to implement MobileNetv2. Another design alteration was deciding to benchmark our design solution against the PYNQ board's ARM SoC CPU instead of an x86 CPU. After conversing with Orshansky over our first batch of test results, it became clear that juxtaposing our FPGA inference time with that of a x86 laptop CPU would be an inaccurate comparison of hardware backends. Instead, the ARM SoC is a more accurate CPU to compare against as it has similar clock frequency and memory resources. To put it simply, it wouldn't be informative to compare the performance of a $150 piece of hardware to that of a $1000 server class CPU or GPU. While this shift in testing would mean starting over part of our test and evaluation process, we felt confident that there would be greater clarity in how our final results should be interpreted and the progression of our project as a whole.

5.0 TEST AND EVALUATION

To test our system, we went through an iterative approach, implementing a new block of our system after each successive test. First in the FrontEnd Framework Test we evaluated TVM's compilation and runtime on the SoC CPU. This gave us a standard to compare each additional optimization against. The VTA Test followed the initial test of the SoC by testing the performance of the SoC and VTA working together. Then, with the AutoTune Test we employed AutoTVM to optimize the CNN for our hardware, and then employed those optimizations to gather acceleration data. The PYNQ Resources Evaluation is unique in that we examined the allocation and use of our board's hardware components. This data gave us an understanding of the flaws in our bitstream or architecture. Finally, in the Hardware Optimization Test we altered the bitstream being used to program the board and recorded inference time to determine how changing the clock speed affects the system's performance. These tests make up the research we have done on our accelerator and give us all the information we need to assess and improve it. All of our data is included in Tables 5 and 6 in Appendix A.

5.1 FrontEnd Framework Test

The FrontEnd Framework Test generalized the initial development steps of our project. These first steps included importing CNN models (ResNet-18, ResNet-50) from the frontend DL framework MxNet, compiling them with TVM, and deploying to the PYNQ board's ARM CPU. This test was a baseline for the performance of our system, as it is the foundation of the rest of the system.

5.1.1 Method

In order to test the different performance metrics of our system, we needed a script that allowed us to adjust the different pieces of our system individually, so we could test the entire system. To accomplish this, our script worked as follows: The ResNet model was imported from a frontend framework and converted into a high level graph through Relay. We quantized the ResNet model from 'float32' operations into operations with the 'int8' data type preferable for VTA. After this the quantized ResNet model is graph packed and offloaded to the specific hardware backend, specified within the script itself. From this step we utilized TVM's Python API to retrieve execution statistics collected after each batch of inferences.

To generate a baseline for the performance of our system and establish a method of testing the execution of the fully optimized system, we ran inference on five different images. These five images can be found in Appendix B. Inference was run on each image and inference time was recorded. In the case of our baseline, inference was executed on the PYNQ board's SoC CPU using ResNet-18 and ResNet-50.

5.1.2 Results and Analysis

As shown in Table 1 on the next page, ResNet-18 has an average inference time of about 1553 ms and ResNet-50 has an average inference time of about 5217 ms. The images referenced are found in Appendix A. On average, ResNet-50 took around 3663 ms longer than ResNet-18. This massive difference is likely due to the difference in size of the two CNNs. ResNet-50 contains significantly more computations than ResNet-18. As mentioned earlier, these numbers are a useful baseline comparison for the data shown in our later tests.

Table 1. FrontEnd Framework Test

Image	Inference Time (ms)	
	ResNet-18	**ResNet-50**
Cat	1552.17	5249.30
Rabbit	1552.40	5204.83
Wheaten Terrier	1555.49	5208.00
Kelpie	1553.04	5215.91
Bernese Mountain Dog	1554.16	5205.88
Average	**1553.45**	**5216.78**

5.2 VTA Test

In the test described in Section 5.1, we tested inference time of the different ResNet CNNs on the PYNQ SoC CPU. In this test, we added in VTA to offload computations from the board's CPU to the FPGA. The addition of this architecture led to decreased inference times on images, as a large portion of the computations were accelerated using the FPGA. This test was compared against the Autotune Test and Hardware Optimization Test to check for improvements on the performance of VTA.

5.2.1 Method

To quantify the improvements brought about by VTA, we used a similar method to the test described in Section 5.1.1. We used the same script as described earlier, with the specific hardware backend set to the PYNQ board. In the same script, VTA was deployed and enabled on the board as well. This allowed for some computation to be offloaded to the FPGA from the CPU itself. Inference was run using the same five images as in the previous two sections and was tested using ResNet-18 and ResNet-50.

5.2.2 Results and Analysis

As shown in Figure 4 below, ResNet-18 had an average inference time of about 765 ms and ResNet-50 had an average inference time of about 1083 ms. The images referenced are found in Appendix B. On average, ResNet-50 took around 319 ms longer than ResNet-18.

It is clear that as predicted, use of the VTA architecture improves inference time significantly when compared to solely the PYNQ SoC CPU. On average, the VTA Test's inference time results were about 789 ms and 4133 ms faster than those of the FrontEnd Framework Test for ResNet-18 and ResNet-50 respectively. This was due to the offloading of computation from the CPU to the FPGA. In the case of ResNet-50, there were more computational tasks that could be offloaded and sped up on the FPGA, which allowed for an even larger performance increase in inference time for ResNet-50 than for that of ResNet-18. When looking at the graph below please keep in mind that "SoC CPU" refers to the PYNQ board's dual-core ARM A9-Cortex processors.

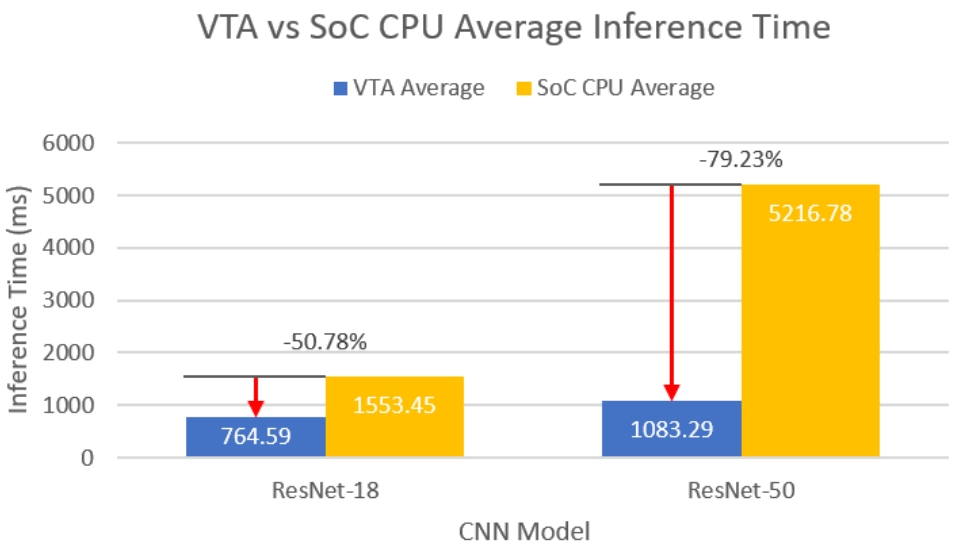

Figure 4. VTA Test Results

5.3 AutoTune Test

Autotuning is one of the optimization steps for the TVM compiler. Autotuning optimizes the graph by altering each unique convolutional operator. The autotuner for TVM is called AutoTVM and relies on an algorithm that defines the way AutoTVM conducts the search for optimizations. The four algorithms we tested for AutoTVM were: GridSearch, RandomSearch, Genetic, and XGB. We autotuned the PYNQ board's CPU with VTA enabled. Then we ran the same script as the previous test, but with our new AutoTVM optimizations enabled.

5.3.1 Method

In order to test the full extent of the compiler's capabilities, we had to test the autotuner. The autotuner comes with different tuning options, so we decided to test four of these - the RandomSearch, Genetic, XGB, and GridSearch algorithms. Each of these algorithms was used to tune the 22 unique convolutions in ResNet-50 for the PYNQ board with VTA enabled. Each tuner algorithm attempted 400 different configurations for each of the 22 convolutions and picked the specific configuration that led to the best performance. These were then placed in a log file that was used by the same script as used in each of the previous tests. Each of the same five images were tested for inference time again, using each of the four different optimized log files.

5.3.2 Results and Analysis

The results of the autotuning tests are shown in Table 2, on the next page. The averages of the inference times using autotuned operations tuned through Random, Grid Search, Genetic Algorithm, and XGB were 1112.36 ms, 1152.50 ms, 1094.68 ms, and 1075.30 ms respectively. Only XGB led to a performance increase on the default configuration for VTA. XGB led to an almost 8 ms decrease in inference time. This performance increase is noticeable, but not significant. The configuration generated by the Genetic Algorithm did the second best of the autotuned configurations, though it did not outperform the default configuration for the board.

Table 2. AutoTune Test Results

Hardware	CNN	AutoTuner	Average (ms)
SoC	ResNet-18	Default	**1553.45**
SoC	ResNet-50	Default	**5216.78**
SoC + VTA	ResNet-18	Default	**764.59**
SoC + VTA	ResNet-50	Default	**1083.29**
SoC + VTA	ResNet-50	Random	**1112.36**
SoC + VTA	ResNet-50	Grid Search	**1152.5**
SoC + VTA	ResNet-50	Genetic Algorithm	**1094.68**
SoC + VTA	ResNet-50	XGB	**1075.3**

5.4 PYNQ Resources Evaluation

In order to examine possible hardware optimizations and gain a deeper understanding of the FPGA's performance, we generated synthesis reports to evaluate the FPGA resources being utilized by the VTA architecture.

5.4.1 Method

To accomplish this we installed Vivado for Linux and regenerated the bitstream through Vivado High-Level Synthesis. These changes included altering the standard makefile that handles the build directory and starting Vivado HLS. Both the Tcl script, which instantiates and connects the VTA architecture block diagram, and the design JSON file, which specifies the input parameters that the Tcl script bases the design off of, had to be updated to generate new VTA bitstreams. One of the parameters in the design JSON file we adjusted was the clock period, calculated using (1), where T denotes the period and f denotes the frequency.

$$T = \frac{1}{f} \tag{1}$$

For our increased clock speed of 142 Megahertz, our calculated frequency was 7 nanoseconds. The faster the clock increases, the more aggressively our design is pipelined. In other words for each clock cycle there is the same amount of work to achieve but less time to accomplish it. Since we are not adding more stages to the pipeline to lessen the workload during each clock cycle, there is a limit to how high we can increase the clock speed before encountering timing violations.

5.4.2 Results and Analysis

Synthesis reports document how the FPGA's four major components, BRAM, Digital Signal Processing (DSP) slices, Flip-Flops (FF), and LUTs, are being utilized. Table 3 below describes the resource summary resulting from generating the standard VTA bitstream clocked at 100 MHz.

Table 3. PYNQ Board Utilization Summary

Module	BRAM_18K	DSP48E	FF	LUT
Compute	77	138	14995	30709
Store	4	0	1481	2465
Load	4	0	4241	18146
Fetch	12	0	972	1189
Total	97	138	21689	52509
Available	**280**	**220**	**106400**	**53200**
Utilization	**34.6%**	**62.7%**	**20.4%**	**98.7%**

Through analyzing this utilization summary, we saw that the majority of the LUTs and DSP slices on the PYNQ board are being instantiated in the compute module. This high resource allocation aligns with our understanding of the VTA architecture as the compute module contains

the General Matrix Multiplication core and Arithmetic Logic Unit, executing high-intensity computations for our neural network. The VTA design overall displays an extremely high LUT utilization which is worrisome, as it is possible that the logic routing becomes congested and it becomes harder to meet timing requirements for the design. High LUT utilization also means that it is difficult to add new features to the design, although this is not a concern for our project goals.

5.5 Hardware Optimization Test

In previous tests, we tested the performance of our system using different optimizations. Conversely, here we tested performance using an adjusted bitstream. We changed the VTA clock frequency to run at 142 MHz, rather than the 100 MHz that it ran at for our previous VTA tests. This allowed us to see if the architecture itself could be improved upon with a faster clock.

5.5.1 Method

We used almost the exact same method as the previous tests to examine the changed bitstream. In this case, we only tested ResNet-50 on the same five images using the new bitstream. The same script used in previous tests was run, but with the altered bitstream rather than the default bitstream. The default bitstream was used in Section 5.3, so the performance changes were determined in reference to the ResNet-50 performance times in the results of that section.

5.5.2 Results & Analysis

Running inference on the five images with ResNet-50 resulted in an average inference time of 1083.02 ms. The five separate inference times with the 142 MHz clock are seen in Table 4, on the next page. Comparing results of the images processed with the 142 MHz clock to the results of the 100 MHz clock shows no improvement. We anticipated a comparatively lower inference time than that of the 100 MHz average of 1083.29 ms but the result was 1083.02 ms on the 142 MHz clock. This was only a difference of 0.27 ms which is not very significant. This slight decrease tells us that speeding up the clock speed of the PYNQ board does not affect inference speed enough to warrant more interest by itself.

Table 4. Hardware Optimization Test Results

Image	Inference Time (ms)
Cat	1081.42
Rabbit	1086.01
Wheaten Terrier	1085.29
Kelpie	1081.19
Bernese Mountain Dog	1081.21
Average	**1083.02**
Performance Change	**+0.27**

6.0 TIME AND COST CONSIDERATIONS

Throughout the course of the project, our team remained well within our budget constraints; however, we faced unexpected time costs due to bugs in TVM's code and the large runtime required to tune our CNN models with AutoTVM. Our only required monetary expenses for our project was a $150 PYNQ board ordered from Digilent through Dr. Orhansky's $500 budget. We also received an additional PYNQ board loaned from us by Dr. McDermott which greatly expanded our team's work throughput. In terms of time considerations, we ran into an issue when trying to set up our RPC server that ended up consuming a large portion of our time. The issue boiled down to errors in TVM's version control (a common problem in open-source projects). After commenting in a TVM forum about the same problem, we eventually formed a solution. Essentially, we were forced to repeatedly remake the TVM runtime on the board to overcome this issue. The developers of the project are currently working to resolve the issue in the next update. As previously mentioned in section 4.0, we ran into other issues with the RPC server that were also caused by bugs in the TVM code, which we ended up being able to resolve ourselves after considerable time expended.

Another large portion of time was spent testing AutoTVM. We realized that tuning the convolutional layers for either ResNet-18 and ResNet-50 was a whole day process that required up to 20 hours to complete. Ultimately this meant that both our available PYNQ boards were busy for long periods of time, so any other work could only progress through the VTA simulator. This slowed down our data collection and often the tuning would fail and the process would need to be restarted. Additionally, AutoTVM requires most of the computing power of a PC, forcing the user off the computer during its multiple hour long or day long run. After all of that, if an issue popped up (such as a brief network error) an entire night of tuning was wasted.

7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN

The main ethical considerations for this design are equivalent to those of Machine Learning (ML). ML has the capability to further many fields and could be extremely beneficial or detrimental depending on the application. No matter what, ML will further the ability of technology to recognize patterns, predict events, and speed up computation. An important consideration of ML is that it lacks common sense and reason, explicitly following the provided logical guidelines. ML carries out the training specified, but often that training is filled with human biases. Even if the training lacks bias, training conducted on a biased sample will often output biased results. There is much debate and concern about this, especially when it comes to facial recognition technology. Often, facial recognition software is trained using data sets not representative of target demographics. This has led to multiple discussions of discrimination regarding the subject [8] . ML is subject to bias and could theoretically do much harm if the algorithm is presumed to be unbiased and, subsequently, not monitored. We considered these points when selecting our own data and frameworks. Throughout this project, the team has consciously worked to prevent such biased research.

8.0 RECOMMENDATIONS

The clear next step for this project would be to modify the source code for the VTA architecture in order to take advantage of more BRAM, minimize LUT utilization, and decrease inference time overall. In the study of Architecture this translates to having more data transfers through DMA instead of traditionally slower accesses from DRAM. Another area of future work is enhancing the optimizations done by TVM scheduler and improved upon by AutoTVM. Part of

the TVM compilation process includes defining and organizing convolutional tasks into loops of code to reduce off-chip accesses through loop tiling and loop partitioning techniques. The parameters necessary for this latency hiding and memory management make up the search space for AutoTVM. Further testing can also be done with other boards and different CNN models. Testing other hardware backends such as GPUs and embedded processors will also characterize TVM's flexibility as an open-source deep learning compiler. In order to test more CNN models we would also need to improve one area that TVM is certainly lacking: quantization support for a wide variety of neural networks. These recommendations range from the compiler side to the circuit intrinsics of the FPGA and only begin to delve into some of the popular graduate-level research topics in the fields of neural networks, compilers, and hardware acceleration.

9.0 CONCLUSION

Our testing confirmed that TVM is a viable DL compiler and that our FPGA-based accelerator significantly improved inference time. Additional testing also confirmed that the transformations done by TVM and VTA did not reduce the model's accuracy when compared with the accuracy rates listed on MxNet. We found that while AutoTVM did lead to decreased inference time, there was little variation in performance when utilizing different tuning algorithms. VTA was also proven to perform the same with a boosted clock frequency. The high LUT and low BRAM usage is an area for further improvement and study of the FPGA hardware accelerator.

While our findings surrounding the TVM compiler were significant enough to cover all the specifications requested, there are still more steps that can be taken to get a better understanding of the compiler. As described in Section 8.0, more testing should be done on different boards and models to better understand the performance enhancement that is specific to the PYNQ board. Expansion of testing would be a natural next step for the project and would allow more insight into how the compiler works.

Our project overall did what it set out to do, testing the efficacy of TVM and VTA to run a CNN for image classification on a PYNQ board. Our team has gained a deeper understanding of deep learning concepts which helped guide us through designing and implementing a solution. The tests we completed aligned with our goals, as they allowed us to see the true performance of

TVM and VTA. While we did run into some problems in regards to the RPC server and building TVM itself, we were able to overcome these obstacles as a team and move forward with the project. We hope that other teams in the future find our work useful and build on what that we have completed over the last year.

REFERENCES

[1] L. Hardesty. (2017, April 14). *Explained: Neural networks* [Online]. Available: http://news.mit.edu/2017/explained-neural-networks-deep-learning-0414.

[2] F. Vásquez. (2017, December 21). *Deep Learning made easy with Deep Cognition* [Online]. Available: https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351.

[3] Intel. *Intel FPGAS Resource Center* [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/new-to-fpgas/resource-center/overview.html.

[4] A. Shawahna, S. M. Sait and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," *IEEE Access,* vol. 7, pp. 7823-7859, 2019.

[5] Apache MXNET. *Predict with pre-trained models* [Online]. Available: https://mxnet.apache.org/versions/1.0.0/tutorials/python/predict_image.html

[6] T. Chen *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," USENIX Assoc., Carlsbad, CA, Rep. 2018.

[7] T .Moreau *et al.*, "A Hardware-Software Blueprint for Flexible Deep Learning Specialization," Dept. Comp. Science & Eng, Univ. Washington, Seattle, WA, Rep. arXiv:1807.04188, 2019.

[8] L. Hardesty. (2018, February 18). *Study finds gender and skin-type bias in commercial artificial-intelligence systems* [Online]. Available: https://news.mit.edu/2018/study-finds-gender-skin-type-bias-artificial-intelligence-systems-0212

[9] T. Moreau. (2018, March 22). *cat.jpg* [Online]. Available: https://homes.cs.washington.edu /~moreau/media/vta/cat.jpg

APPENDIX A – TESTING TIMING DATA

APPENDIX A – TESTING TIMING DATA

Table 5. Inference Timing Test Results (ResNet18)

Hardware	Cat (ms)	Rabbit (ms)	Wheaten Terrier (ms)	Kelpie (ms)	Bernese Mountain Dog (ms)	Average (ms)
SoC	1552.17	1552.4	1555.49	1553.04	1554.16	*1553.45*
SoC + VTA	764.43	764.01	765.58	764.58	764.33	*764.59*

Table 6. Inference Timing Test Results (ResNet50)

Hardware	Auto-Tuner	Clock Speed (MHz)	Cat (ms)	Rabbit (ms)	Wheaten Terrier (ms)	Kelpie (ms)	Bernese Mountain Dog (ms)	Average (ms)
SoC	None	N/A	5249.3	5204.83	5208	5215.91	5205.88	*5216.78*
SoC + VTA	None	100	1086.77	1083.58	1085.56	1080.34	1080.22	*1083.29*
SoC + VTA	Random	100	1112.47	1112.35	1112.48	1112.18	1112.33	*1112.36*
SoC + VTA	Grid Search	100	1155.85	1156.63	1146.64	1151.92	1151.47	*1152.5*
SoC + VTA	Genetic Algorithm	100	1096.91	1091.31	1091.79	1102.07	1091.36	*1094.68*
SoC + VTA	XGB	100	1072.46	1074.25	1072.4	1081.21	1076.2	*1075.3*
SoC + VTA	None	142	1081.42	1086.01	1085.29	1081.19	1081.21	*1083.02*

APPENDIX B – IMAGES USED IN INFERENCE

APPENDIX B – IMAGES USED IN INFERENCE

Figure 5. Cat [9]

Figure 6. Rabbit

Figure 7. Wheaten Terrier

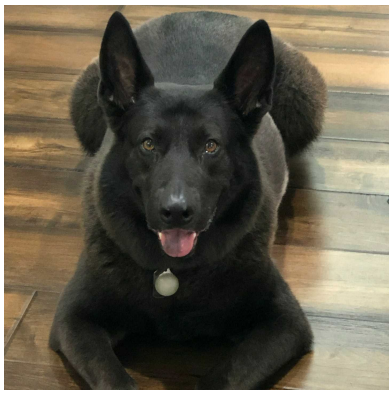

Figure 8. Kelpie

Figure 9. Bernese Mountain Dog

APPENDIX C – BILL OF MATERIALS

APPENDIX C – BILL OF MATERIALS

Table 7. Bill of Materials

Material	Price	Description
PYNQ-ZI Board	$147.50	FPGA, primary hardware used to implement the project
PYNQ-Z2 Board	$0.00	FPGA, given to us by Prof. McDermott
MicroSD Card	$0.00	Used with FPGA, given to us by Prof. McDermott
Total	**$147.50**	**All Costs**