# TVM a Machine Learning Compiler and Optimizer

*Michael Sampson*
*Computer Science Department, University of Crete*

## Abstract

In today's era, the surge in machine learning has prompted companies to design specialized software to meet the constant demand posed by emerging hardware. While this approach works for some, it often requires a substantial amount of human resources, leading to inefficiencies due to the lack of automation. In this context, I present Apache's TVM, an open-source deep-learning compiler framework designed to optimize and run computations efficiently on diverse hardware backends. This project demonstrates TVM's capability for automatic hardware tuning (AutoTVM) using CUDA (NVIDIA's GPU backend) and LLVM, I analyzed results against a PyTorch baseline to highlight TVM's efficiency.

In this thesis, I will guide users through deploying TVM and maximizing AutoTVM's tuning capabilities, detailing installation and configuration steps. I will showcase comprehensive settings and options for achieving optimal tuning results and delve into the auto-tuning process, including task extraction, building, and running tasks. By leveraging TVM's automated tuning capabilities, we emphasize its performance optimization advantages across different hardware backends and models.

AutoTVM allows for specifying neural network layers, prompting an investigation into comparing the results of specifying 2D convolutions versus tuning across all neural network layers. To explore this, I utilized Vision Transformers (ViT) as a sophisticated model. Additionally, I employed Bidirectional Encoder Representations from Transformers (BERT) to evaluate tuning effectiveness in a complex NLP model. Furthermore, I implemented a database to store tuning configurations tailored to model type, batch size, and device, enabling efficient execution with the most optimized TVM setup.

This project not only explores how TVM can revolutionize deep learning optimizations but also introduces practical implementations to streamline model tuning and deployment. Emphasizing TVM's role in advancing performance optimization in deep learning, this work contributes to the broader field of machine learning infrastructure.

## 1 Introduction

### 1.1 Problem Statement

The core problem addressed in this thesis is the inefficiency and resource-intensive nature of manual tuning processes for optimizing hardware backends in machine learning (ML) applications. Currently, achieving optimal performance on diverse hardware configurations requires extensive manual effort, including code refactoring, algorithm optimization, and specialized tuning by skilled personnel. These processes are costly in terms of time, financial resources, and human capital. There is a pressing need for automated solutions that can streamline and consolidate these optimization tasks.

Apache TVM offers AutoTVM as a solution to automate the tuning of machine learning models for specific hardware configurations. By leveraging AutoTVM, TVM aims to reduce the dependency on manual intervention in optimizing code elements tailored to hardware specifics. This approach significantly improves efficiency in achieving high-performance ML applications across various hardware platforms.

### 1.2 Contributions

The objectives of this thesis are to evaluate the efficacy of Apache TVM with AutoTVM compared to traditional optimization backends like PyTorch, specifically focusing on image classification tasks using models such as ResNet-18. Additionally, the research investigates whether TVM can achieve superior results by optimizing more complex models with diverse neural network layers beyond standard convolutions (conv2d). Furthermore, the thesis aims to develop a comprehensive automated database that captures and utilizes the best AutoTVM tuning configurations tailored to different ML models and hardware backends.

- **Objective 1:** Set up and execute TVM within Jupyter Notebooks.
- **Objective 2:** Evaluate Auto-Generated Operator Code in TVM against Highly Optimized Libraries such as CUDNN.
  - **Sub-objective 1:** Conduct image classification using ResNet-18 in PyTorch.
  - **Sub-objective 2:** Conduct image classification using ResNet-18 with TVM and AutoTVM for inference.
  - **Sub-objective 3:** Optimize with AutoTVM and compare timing results with PyTorch.
- **Objective 3:** Compare neural network operators, focusing on conv2d.
  - **Sub-objective 1:** Analyze results on ResNet-18.
  - **Sub-objective 2:** Analyze results on more diverse models.
- **Objective 4:** Develop an automated database containing the best AutoTVM tuning configurations for each model and backend.

## 2 Theoretical Background

In this section, the theoretical foundations relevant to optimization techniques and methodologies in deep learning are explored, with a specific focus on TVM and automated tuning. The following aspects are addressed within the framework of the experiment:

### 2.1 Fundamental Concepts in AI and Deep Learning

Artificial intelligence (AI) is a rapidly evolving field that aims to enable machines to perform tasks that traditionally require human intelligence. It encompasses a wide range of methodologies and techniques that have found applications across various domains, revolutionizing modern computing.

### 2.2 Introduction to Machine Learning and Understanding Deep Learning

My journey into AI began with a comprehensive introduction to essential tools and algorithms in machine learning. I gained proficiency in foundational supervised learning techniques such as linear and logistic regression, which are crucial for tasks involving prediction and classification.

Expanding my knowledge, I delved into advanced neural network architectures during my studies. This included practical implementations of convolutional neural networks (CNNs) for image classification and object detection. Additionally, I explored neural style transfer techniques, the application of recurrent neural networks (RNNs) for sequential data analysis and language modeling, and transformer architectures for advanced natural language processing tasks like text generation and sentiment analysis.

Throughout my exploration, I focused on optimizing deep learning models using techniques such as regularization, gradient descent variants, and hyperparameter tuning. These concepts form the bedrock of my research into advanced topics in deep learning optimization, particularly in the context of leveraging automated tuning techniques with TVM for my thesis work.

### 2.3 Overview of TVM's Significance

TVM, which stands for Tensor Virtual Machine, is an end-to-end compiler stack for deep learning that provides a comprehensive set of optimizations for a wide variety of hardware backends. Unlike other existing frameworks, TVM prioritizes both graph-level and operator-level optimizations, ensuring high performance across diverse hardware backends. In recent years, the diversity in on-chip memory architecture necessitates the generation of optimized code to effectively utilize the hardware capabilities. Traditional deep learning (DL) frameworks such as TensorFlow and PyTorch require manual tuning due to high-level graph optimizations, which are often insufficient for specific hardware back-ends. TVM, an optimizing compiler for deep learning, addresses this by taking high-level specifications of DL programs from existing frameworks and generating low-level optimized code for a diverse set of hardware back-ends. To achieve competitive performance, TVM tackles several key challenges. The next sections will delve into these key features in detail, explaining how TVM achieves its remarkable efficiency and flexibility.

- **DL Accelerators and Hardware Primitives:** Optimizing code for DL accelerators, GPUs, and CPUs requires a deep understanding and utilization of specialized compute primitives. TVM leverages these primitives to generate efficient code tailored to the target hardware.
- **Complex Inputs and Memory Hierarchies:** Efficient memory management is crucial for optimizing performance due to the diverse input types and memory structures of different hardware. TVM addresses this by optimizing memory access patterns and data layouts.
- **Large Search Space for Optimization:** Generating efficient code involves exploring numerous configuration options, necessitating a balance between search costs and performance gains. TVM employs machine learning-based cost models to navigate this large search space effectively.

TVM addresses these challenges through several innovative approaches:

- **Introducing a Tensor Expression Language:** TVM provides program transformation primitives that generate different versions of the program with various optimizations using a tensor expression language.
- **Automated Program Optimization Framework:** TVM uses machine learning techniques to automatically find optimized tensor operators by collecting and analyzing performance data.
- **Graph Rewriter:** TVM utilizes a graph rewriter that applies both high-level and operator-level optimizations to further enhance performance.

## 2.4 TVM Architecture and Components

TVM's approach to optimizing DL models involves several key components and methodologies [5]:

### 2.4.1 Optimizing Computational Graphs

Computational graphs represent programs in DL frameworks, illustrating operations and data dependencies using large, multidimensional tensors. TVM leverages this representation for high-level optimizations:

- **Operator Fusion:** Operator fusion combines multiple small operations into a single kernel, avoiding intermediate memory storage. TVM categorizes graph operators into four types: injective (e.g., add), reduction (e.g., sum), complex-out-fusable (e.g., conv2d), and opaque (e.g., sort). For instance, conv2d operations can fuse element-wise operators to their output, significantly improving execution time by 1.2× to 2× across various workloads.
- **Data Layout Transformations:** Data layout transformations optimize internal data formats for better hardware compatibility. Common data layouts include column-major and row-major. TVM converts computational graphs to utilize optimal internal data layouts, enhancing execution efficiency on target hardware.

### 2.4.2 Generating Tensor Operations

TVM generates efficient code for each operator by producing multiple valid implementations on each hardware back-end and selecting the optimized version:

- **Tensor Expression Language** The Tensor Expression (TE) language within TVM is a highly specialized domain-specific language designed for the efficient description of tensor computations. Operating in a purely functional manner, TE facilitates easier reasoning and optimization of tensor operations by transforming input tensors into output tensors, thus enabling precise control over the computation process. By combining TE expressions with a corresponding schedule, executable code can be generated for specific target languages and architectures, such as LLVM and a CPU. This process involves providing TVM with the schedule, the list of TE expressions included in the schedule, the target and host information, and the name of the function to be generated, resulting in a type-erased function that can be directly invoked from Python.
- **Tensor Expression and Schedule Space:** TVM's tensor expression and schedule space allow flexible computation expressions. It separates computation from scheduling, similar to Halide, enabling efficient mappings to different hardware. Schedule primitives define transformations that preserve program logic, crucial for optimizing performance across various hardware platforms.
- **Nested Parallelism with Cooperation:** TVM introduces cooperative data fetching, allowing GPU threads to share memory and enhance parallelism and data reuse. This is particularly beneficial in operations like matrix multiplication.
- **Tensorization:** Tensorization decomposes computations into tensor operators (e.g., matrix-matrix multiplication, 1D convolution). TVM uses tensor-intrinsic declarations and tensorize techniques to optimize computations for different hardware, simplifying the process and improving execution efficiency.
- **Explicit Memory Latency Hiding:** TVM optimizes memory and compute resource utilization by overlapping memory operations with computation. Techniques vary by hardware: simultaneous multithreading for CPUs, rapid context switching for GPUs, and a decoupled access-execute (DAE) pipeline for specialized DL accelerators. TVM introduces virtual threading scheduling primitives to automate low-level synchronization, ensuring correct execution and efficient pipeline parallelism.

### 2.4.3 Automating Optimization

TVM aims to find the optimal implementation for each DL model layer by customizing operators for specific data shapes and layouts. This process involves several steps:

- **Schedule Space Specification:** TVM provides an API for specifying task-specific handling methods, along with default templates for various hardware, aiding the optimizer in selecting the best configuration.
- **ML-Based Cost Model:** TVM employs a machine learning-based cost model to predict the performance of

different schedule tasks for DL. This model balances the need for extensive experimentation with the ability to make informed predictions.

- **Auto-Tuning:** Auto-tuning involves exploring numerous configurations to find the best one. TVM combines predefined cost models and ML predictions to balance exploration time and performance gains.
- **Schedule Exploration:** TVM uses a simulated annealing algorithm to explore different configurations, adjusting them based on ML model predictions. Successful adjustments improve predicted performance, leading to better configurations.
- **Distributed Device Pool and RPC:** TVM's customized RPC system enables remote compilation and execution on specific devices, facilitating resource sharing and automating cross-compilation and performance measurement. This infrastructure streamlines optimization processes, making them feasible for embedded devices requiring manual effort for code deployment and testing.

In summary, TVM addresses the complexities of optimizing deep learning models for diverse hardware back-ends by leveraging a combination of tensor expression language, automated program optimization, graph rewriting, and machine learning-based cost models. This comprehensive approach allows TVM to generate highly optimized code, enhancing performance across various hardware platforms.

## 2.5 Learning to Optimize Tensor Programs

Deep learning (DL) heavily relies on tensor operators like matrix multiplication and convolution. Optimizing these for diverse hardware platforms is challenging due to the multitude of implementation choices. In order to address this challenge with machine learning to automatically optimize tensor operator programs [6]. Contributions include:

- Formalizing the problem of learning to optimize tensor programs.
- Developing a machine learning framework for automatic optimization.
- Achieving 2x to 10x acceleration in optimization using transfer learning.

The problem involves specifying tensor operators using index expressions and generating multiple low-level code variants. The goal is to minimize runtime cost on hardware, determined through experiments. Key prerequisites include defining an exhaustive search space for hardware-aware optimizations and efficiently finding optimal schedules within this space. The framework includes:

- **Expression Language:** Specifies tensor operations.
- **Exploration Module:** Manages search loop based on cost estimates.
- **Statistical Cost Model:** Predicts runtime costs of programs.
- **Code Generator:** Generates optimized low-level code.
- **Hardware Environment:** Executes experiments on real hardware.

### 2.5.1 Learning to Optimize Tensor Programs

We propose a machine learning-based framework:

- **Statistical Cost Model:** Estimates runtime cost $f(x)$ of a program $x$ using Gradient Boosted Trees (GBTs) and TreeGRU. GBTs focus on precise feature extraction from an AST, while TreeGRU recursively encodes the AST into an embedding vector, leveraging batching and GPU acceleration for efficient training and prediction.
- **Training Objective Function:** Utilizes regression or rank loss functions to optimize the statistical cost model using a dataset $D = \{(e_i, s_i, c_i)\}$, where $e_i$ denotes expressions, $s_i$ represents schedules, and $c_i$ indicates computational costs.
- **Exploration Module:** Employs simulated annealing with $f(x)$ as an energy function to navigate the program search space, integrating diversity-aware exploration to enhance the quality and robustness of candidate solutions.

### 2.5.2 Accelerating Optimization via Transfer Learning

Transfer learning speeds up optimization across different tasks by leveraging knowledge gained from previous tasks or domains. In the context of optimizing tensor programs, transfer learning involves:

- **Transferable Representation:** The aim to create a representation that works across different tasks. Instead of directly using the configuration of programs (which can change), we focus on the low-level loop structure, which remains consistent.
- **Context Relation Features:** Features at each loop level to capture loop characteristics. These features are enhanced with context relation features, which model relationships between different loop attributes. This helps in understanding how different loop characteristics affect performance.

- **Context Encoded TreeGRU:** Encode loop structures using a neural-based model called TreeGRU. However, as loop variables can vary across tasks, we encode each loop variable using context vectors extracted from our previous features. This ensures consistency across different tasks.
- **Transfer Learning Method:** Combines a global model trained on historical data with a local model trained specifically for the current task. This hybrid approach helps in making effective initial predictions, even when there's limited data available for the current task.

### 2.5.3 Prior Work

Prior methods include black-box optimization and hardware-dependent cost models. This approach uses statistical cost models inspired by SAT solvers, complementing recent trends in deep neural networks for program analysis.

## 2.6 Information Background for Tasks and Tuning

The following background details are essential for understanding the functioning of the tasks. This section provides additional information as foundational knowledge, which will be elaborated upon in the methodology section.

**Sample Configurations** Here are some example configurations that the tuner might explore:

- **Config 1**: `tile f`
- **Config 2**: `tile y`
- **Config 3**: `tile x`
- **Config 4**: `tile rc`
- **Config 5**: `tile ry`

Each of these configurations represents a different way to break down the task into smaller, more manageable chunks for tuning. For example:

- **tile f**: Tiling factor for the filter dimension.
- **tile y**: Tiling factor for the output height dimension.
- **tile x**: Tiling factor for the output width dimension.
- **tile rc**: Tiling factor for the reduction channel dimension.
- **tile ry**: Tiling factor for the reduction height dimension.

These configurations allow the tuner to explore different strategies for executing the task and find the most efficient one for the target hardware.

**Configuration Space and Logic** The configuration space defines all possible values for the tunable parameters. This space is typically vast, containing millions of possible configurations. The tuner employs various search algorithms to navigate this space efficiently.

**Example Configuration Logic**:

- **Tiling**: Dividing the computation into smaller tiles to optimize for cache usage.
- **Loop Unrolling**: Unrolling loops to increase instruction-level parallelism.
- **Parallelization**: Distributing computation across multiple threads or cores.
- **Vectorization**: Using vector instructions to perform multiple operations in parallel.

By iterating through the configuration space and testing different configurations, the tuner aims to find the optimal set of parameters that maximizes performance on the target hardware.

**Layout Format** Data layout format describes how multi-dimensional data is organized and stored in memory [10]. This organization can significantly impact the performance of neural network operations by affecting spatial and temporal locality. In this document, we explain common data layout formats used in neural networks, their representation, and their effects.

**Common Data Layout Formats**

- **1D Data Layouts**
  - **NCW**: This format is used for 1-dimensional data with three dimensions:
    - **N**: Batch size - the number of data instances in a batch.
    - **C**: Channels - the number of feature maps or filters.
    - **W**: Width - the spatial dimension.
  - **NWC**: Another 1D data layout with dimensions in the order:
    - **N**: Batch size.
    - **W**: Width.
    - **C**: Channels.

- **2D Data Layouts**
  - **NCHW**: This format is used for 2-dimensional data with four dimensions:
    - **N**: Batch size.
    - **C**: Channels.
    - **H**: Height - the vertical dimension.
    - **W**: Width - the horizontal dimension.
  - **NHWC**: Another 2D data layout with dimensions in the order:
    - **N**: Batch size.
    - **H**: Height.
    - **W**: Width.
    - **C**: Channels.
- **3D Data Layouts**
  - **NCDHW**: This format is used for 3-dimensional data with five dimensions:
    - **N**: Batch size.
    - **C**: Channels.
    - **D**: Depth - the third spatial dimension.
    - **H**: Height.
    - **W**: Width.
  - **NDHWC**: Another 3D data layout with dimensions in the order:
    - **N**: Batch size.
    - **D**: Depth.
    - **H**: Height.
    - **W**: Width.
    - **C**: Channels.

**Impact on Performance** Different data layouts can have a significant impact on performance depending on the hardware backend and the operations being performed.

- **Spatial Locality**: Data layout affects how data elements are arranged in memory. Layouts like **NCHW** are often preferred on GPUs because they allow for coalesced memory accesses, improving memory bandwidth utilization.
- **Temporal Locality**: Certain layouts improve cache utilization by ensuring that data needed at similar times is stored close together. For example, **NCHWc**, where the channel dimension is tiled, can exploit data locality more efficiently.
- **Framework Compatibility**: Some frameworks have default data layouts that optimize for the common use cases of that framework. For instance, TensorFlow often uses **NHWC**, while PyTorch uses **NCHW**.

#### 2.6.1 Supported Neural Network Operations

TVM supports a wide range of neural network operations. In Figure 1 is a list of the operators that are supported.

This extensive list demonstrates TVM's capability to support a variety of operations needed for building and optimizing complex neural network models. [8]

#### 2.6.2 XGBTuner

XGBTuner is a tuning algorithm that leverages XGBoost as a cost model. The specifics of the tuning process will be elaborated upon subsequently; here, we will focus on understanding XGBoost. [3]

**Introduction to Boosted Trees**

XGBoost, or "Extreme Gradient Boosting," is an implementation of gradient boosted decision trees designed for speed and performance. Gradient boosting constructs a predictive model as an ensemble of weak predictive models, typically decision trees.

**Elements of Supervised Learning**

XGBoost is used for supervised learning problems, where the goal is to predict a target variable using training data composed of multiple features.

**Model and Parameters**

In supervised learning, the model refers to the mathematical framework used to make predictions from input features. Parameters are the elements within the model that must be learned from the data.

**Objective Function**

The objective function combines the training loss, which measures how well the model predicts the training data, and the regularization term, which penalizes model complexity to prevent overfitting. Common loss functions include mean squared error (MSE) and logistic loss.

**Decision Tree Ensembles**

XGBoost uses an ensemble of Classification and Regression Trees (CART). Each tree contributes to the final prediction, and the ensemble improves predictive performance by combining the predictions of individual trees.

**Tree Boosting**

Tree boosting trains an ensemble of trees sequentially, where each new tree attempts to correct errors made by the previous trees. The prediction at each step is the sum of the predictions from all previous trees.

**Model Complexity**

The complexity of a tree is defined by the number of leaves and the magnitude of the leaf scores. Regularization helps control overfitting by penalizing overly complex trees.

**Training with Additive Strategy**

An additive strategy is used for training, adding one tree at a time to minimize the objective function, which includes both the training loss and the regularization term.

**Optimization and Efficiency**

| Category | Operations |
|---|---|
| Convolutions | `conv1d`<br>`conv2d`<br>`conv3d`<br>`deformable_conv2d`<br>`depthwise_conv2d`<br>`conv1d_transpose`<br>`conv2d_transpose`<br>`conv3d_transpose`<br>`bitserial_conv2d` |
| Pooling | `pooling`<br>`dilate` |
| Normalization | `instance_norm`<br>`layer_norm`<br>`group_norm`<br>`rms_norm`<br>`batch_norm`<br>`local_response_norm` |
| Dense Layers | `dense`<br>`bitserial_dense` |
| Element-wise Operations | `elemwise`<br>`flatten`<br>`pad` |
| Matrix Multiplications | `batch_matmul`<br>`mapping` |
| Upsampling | `upsampling`<br>`depth_to_space`<br>`space_to_depth`<br>`space_to_batch_nd`<br>`batch_to_space_nd` |
| Softmax | `softmax` |
| Correlation | `correlation` |
| Binary Neural Networks | `bnn` |
| Quantized Neural Networks | `qnn` |
| Recurrent Neural Networks | `lstm` |
| Specialized Layers | `sparse`<br>`fifo_buffer` |
| Loss Functions | `loss` |

Figure 1: Neural Network Operators
List of supported neural network operators

XGBoost incorporates several systems optimizations, such as parallelized tree construction and distributed computing capabilities, making it suitable for large-scale datasets.

# 3 Methodology

This chapter outlines the various steps undertaken to execute the practical aspects of the project, aiming to generate data and results while concurrently gathering insights into the workings of TVM and its AutoTVM (autotuning) module. Initially, the goal was to understand the basic functionality of TVM. However, as the project progressed, the objectives evolved to include an in-depth examination of how AutoTVM achieves its optimization results and a comparative analysis of its performance against other machine learning compilers such as PyTorch, focusing on CUDA and LLVM backends. After tuning ResNet, which relies heavily on convolutions, we sought to understand how tuning for a model with little to no convolutions, such as Vision Transformer (ViT), would perform. This involved auto-tuning specifically for convolutions versus tuning for every neural network layer.

Throughout this process, I engaged in extensive review and analysis of documentation and libraries relevant to TVM and AutoTVM. This iterative learning and application process not only broadened my understanding but also refined my goals, ultimately leading to a comprehensive evaluation of TVM's auto-tuning capabilities in the context of modern machine learning workflows.

## 3.1 Installation and Setup

### 3.1.1 Running Jupyter Notebooks Remotely

To enable remote access to Jupyter Notebooks on a Linux machine:

```
$ jupyter notebook --ip=192.111.1.111 --port=8888
```

Exposing the port:

```
$ expose machine-name 8888
```

Ensure continuous operation using tmux if issues arise.

### 3.1.2 Conda Environment

Utilize Conda for managing dependencies and creating isolated environments:

- Conda ensures stable installations of TVM and related packages without system-wide conflicts.
- Facilitates easy switching between projects and dependencies.

### 3.1.3 Image Classification Setup

For image classification tasks:

- Used ResNet-50 for initial ML model coding practice.
- Attempted inference from the MLCommons repository.
- Benchmarked with MLPerf using Docker for performance evaluation.

## 3.2 Summary

This setup aimed to establish a robust environment for conducting machine learning experiments with TVM on a CentOS machine. Key steps included remote Jupyter setup, Conda environment management for dependency control, and initial benchmarks for performance evaluation using MLPerf standards.

## Note

Ensure to generalize any specific IP addresses, usernames, or other sensitive details to maintain confidentiality in your thesis. Focus on summarizing the steps taken and their relevance to your research without delving into unnecessary technical minutiae. This approach will help maintain clarity and conciseness while aligning with the purpose of your thesis.

### 3.2.1 TVM Setup

Installing TVM initially on CentOS posed challenges. After encountering segmentation faults, I resolved the issue by enabling GCC-11 with:

```
scl enable devtoolset-11 bash
```

Subsequently, I attempted TVM installation using Conda environments, but encountered issues. Transitioning to Ubuntu Linux OS, I revisited the setup:

```
cmake -DCMAKE_C_COMPILER=/usr/bin/gcc \
-DCMAKE_CXX_COMPILER=/usr/bin/g++ ..
```

Following successful build configurations, I resolved clock skew warnings. Although initial attempts with ResNet-50 on CPUs failed, GPU configurations performed adequately. Final TVM setup involved resolving CUDA and module import issues, ultimately ensuring correct operation with:

```
pip install --pre mlc-ai-nightly -f
https://mlc.ai/wheels
```

Moving forward, I simplified experiments with PyTorch integration for NVIDIA GPU, which proved more stable during auto-tuning processes. Next I will detail the correct steps for installing TVM on Ubuntu Operating Systems:

1. Clone the git repository:

```
git clone --recursive https://github.com/apache/tvm\
tvm
```

2. Download dependencies:

```
sudo apt-get update
sudo apt-get install -y python3 python3-dev\
python3-setuptools gcc libtinfo-dev zlib1g-dev\
build-essential cmake libedit-dev libxml2-dev
```

3. Open TVM in your conda environment, create a build folder, and copy the configuration:

```
conda activate tvm-env
cd tvm
mkdir build
cp cmake/config.cmake build
cd build
```

4. Edit the config to your needs:

```
nano config.cmake
```

5. Run cmake using the conda prefix:

```
echo $CONDA_PREFIX
cmake .. -DCMAKE_INSTALL_PREFIX=/path/to/conda\
/prefix
# Example:
cmake .. -DCMAKE_INSTALL_PREFIX=/home1/public\
/misampson/miniconda3/envs/tvm
```

6. Make and install the configuration:

```
make -j‘nproc‘ install
```

7. Go back and run the Python setup to install the packages:

```
cd ../python
python setup.py install
```

8. Install dependencies:

```
pip3 install --user numpy decorator attrs\
typing-extensions psutil scipy tornado\
'xgboost>=1.1.0' cloudpickle
```

9. To run the notebooks in your environment:

```
conda install -c conda-forge tensorflow anaconda
ipykernel
python -m ipykernel install --user --name=tvm-env
```

10. Run Jupyter Notebook:

```
jupyter notebook
```

Congrats! Now you can open the Jupyter notebook and use TVM to run models.

## 3.3 Working with TVM

### 3.3.1 Running a Model with TVM

Initially, attempts to run TVM notebooks produced errors due to an incorrect setup. After resolving the setup issues and properly configuring TVM for CUDA, I attempted to use ONNX for model extraction. The initial path for 'resnet50.onnx' was unavailable, and through further investigation, I identified a working ONNX model [4]. By setting the input name and size of the model by extracting this information from the ONNX file, it worked. [12]

### 3.3.2 Maximum Size

With TVM completely working with CUDA, I started with finding the largest batch size I could run at a single time. The upper limit was around 227 for ResNet-50. During this step, I learned that the correct input shape is a prerequisite for any model. For running ResNet-50, the input shape should be `[batch_size, 3, 224, 224]`—the batch size is the number of images processed at the same time, followed by RGB channels, and the dimensions of the image, specific for each model. I also discovered that from an ONNX file, both the input shape and input name can be extracted.

## 3.4 Autotuning Options

Tuning in TVM involves optimizing a model to run faster on a specified target device. The primary goal in autotuning is to configure various parameters, as detailed in the following subsections, to create the tuner object. The auto-tuner identifies tasks to guide the tuning process, aiming to find an optimal configuration for the model to enhance its performance. Unlike training or fine-tuning, tuning focuses solely on improving runtime performance without affecting the model's accuracy. During tuning, TVM tests numerous operator implementation variants to determine the most efficient one, with the results saved in a tuning records file. I experimented with different tuning options to compare their impact on tuning efficiency and runtime.

### 3.4.1 Task Extraction

The most important step in autotuning is producing tasks so that they can later be tuned. The tasks take as arguments the model, the target device, the parameters of the model, and the operations (ops). A task is a tunable composition of template functions.

The tuner takes a tunable task and optimizes the joint configuration space of all the template functions in the task. This module defines the task data structure, as well as a collection (zoo) of typical tasks of interest.

**Definition of Task Function** A task can be constructed from a tuple of `func`, `args`, and `kwargs`. `func` is a stateless function or a string that registers the standard task. For more detailed information, you can refer to the TVM Autotvm Documentation.

**Arguments**

- **Model**: The neural network or machine learning model to be tuned.
- **Target Device**: The hardware device (e.g., CPU, GPU) on which the model will be run.
- **Parameters of the Model**: Hyperparameters and other configuration details specific to the model.
- **Ops (Operations)**: The specific operations (e.g., convolutions, matrix multiplications) that need to be tuned for optimal performance.

**Example of Extracted Task** Here is an example of what an extracted task might look like:

**Extracted Task:**

```
Task 1/1
  Name: conv2d_nchw.cuda
  Args: (('TENSOR', (1, 3, 224, 224), 'float32'),
  ('TENSOR', (64, 3, 7, 7), 'float32'),
  (2, 2), (3, 3, 3, 3), (1, 1), 'float32')
  Workload: ('conv2d_nchw.cuda',
  ('TENSOR', (1, 3, 224, 224), 'float32'),
  ('TENSOR', (64, 3, 7, 7), 'float32'),
  (2, 2), (3, 3, 3, 3), (1, 1), 'float32')
  Config Space Size: 79027200
  Target: cuda -keys=cuda,gpu -arch=sm_75
  -max_num_threads=1024 -model=unknown
  -thread_warp_size=32
  FLOPs: 236027904.0
  Function: <tvm.autotvm.task.task.TaskTemplate
  object at 0x7f141446e5d0>
  KWArgs: {}
  Sample Configurations:
    Config 1: tile_f
    Config 2: tile_y
    Config 3: tile_x
    Config 4: tile_rc
    Config 5: tile_ry
```

**Details of the Extracted Task**
**Name**: `conv2d nchw.cuda`
**Args**:

- `('TENSOR', (1, 3, 224, 224), 'float32')`: The input tensor with shape `(1, 3, 224, 224)` and data type `float32`.
- `('TENSOR', (64, 3, 7, 7), 'float32')`: The kernel tensor with shape `(64, 3, 7, 7)` and data type `float32`.
- `(2, 2)`: The stride of the convolution.
- `(3, 3, 3, 3)`: The padding for the convolution.
- `(1, 1)`: The dilation of the convolution.
- `'float32'`: The data type of the output tensor.

**Workload**: A tuple describing the entire task:

```
('conv2d nchw.cuda',
('TENSOR', (1, 3, 224, 224), 'float32'),
('TENSOR', (64, 3, 7, 7), 'float32'),
(2, 2), (3, 3, 3, 3), (1, 1), 'float32')
```

**Config Space Size**: `79027200`: The size of the configuration space to explore.
**Target**:

```
cuda -keys=cuda,gpu -arch=sm_75
-max_num_threads=1024 -model=unknown
-thread_warp_size=32
```

**FLOPs**: `236027904.0`: The number of floating point operations for the task.
**Function**: `<tvm.autotvm.task.task.TaskTemplate object at 0x7f141446e5d0>`: The function template used for the task.
**KWArgs**: : Any additional keyword arguments.

**Conclusion** Understanding and correctly using data layouts is crucial for optimizing neural network performance. Different layouts may be preferred depending on the specific hardware backend and the neural network architecture. By transforming data layouts appropriately, one can achieve significant performance improvements in both training and inference.

### 3.4.2 Measure Option

The measure options' role is divided between building and running the extracted tasks to fine-tune the model and produce a tuner object. It specifies how each configuration is measured, including the number of runs and timeouts. The tuning process begins by defining tasks and running them through a builder and a runner. The builder compiles the high-level task specifications into executable code, while the runner measures the performance of the compiled code.

**The Builder** The builder is responsible for transforming high-level task specifications into executable code. It takes the following input and output:
**Input:** *MeasureInput* (which includes target, task, and config)
**Output:** *BuildResults*
The builder can be configured to use either an RPC builder, which builds programs on a remote device, or a local builder. In this context, only the local builder was used. The builder primarily takes a timeout parameter and performs the following steps:

- For each *MeasureInput* in *measure inputs*, the method extracts the target device, task, and configuration.
- It then compiles the program using TVM's build functions. This involves creating a schedule, compiling the schedule into a function, and exporting the function as a library (shared object file).
- The time taken to build the program is recorded.

The builder manages parallel build tasks, handles temporary storage of compiled files, and collects *BuildResults*. Compiled files are saved in a temporary directory, ensuring a clean and isolated environment for each build session. Filenames are generated using a random 64-bit number to ensure uniqueness [9]:

```
filename = os.path.join(tmp dir,
f"tmp func {getrandbits(64):0x}.
{self.build func.output format}")
```

**The Runner** The runner executes the compiled code and measures its performance. It can use either an RPC runner, which runs the programs on remote devices, or a local runner. The runner's configuration includes several key parameters:

- **number**: Specifies the number of different configurations to test.
- **repeat**: Specifies how many measurements to take of each configuration.
- **min repeat ms**: Specifies the minimum time to run the configuration test. If the number of repeats falls under this time, it will be increased. This option is necessary for accurate tuning on GPUs and is not required for CPU tuning. Setting this value to 0 disables it.

- **timeout**: Places an upper limit on how long to run the training code for each tested configuration.

**Input:** Same *MeasureInput* (which includes target, task, and config) and *BuildResults*
**Output:** *MeasureResult*

The runner initializes with the necessary parameters, sets up the task and device, submits tasks for execution, collects performance results or errors, and returns a list of *MeasureResult* objects. *BuildResult* is the result returned from the builder, containing the path to the generated library, while *MeasureResult* stores all the results of a measurement.

### 3.4.3 Tuner

TVM and AutoTVM use a tuner to find the best implementation after extracting the tuning data. The method it uses to search for the best implementation is determined by the tuner. TVM offers various automatic tuners that work in different ways:

- **Random Tuner**: Selects configurations randomly from the search space.
- **GA Tuner (Genetic Algorithm)**: This tuner does not have a cost model, so it always runs measurements on real machines. It uses a genetic algorithm to find the best implementation.
- **GridSearch Tuner**: Enumerates the search space in a grid search order.
- **XGBoost Tuner**: Uses XGBoost as a cost model. In Figure 2 are the options for tuning with xgb.

In summary, XGBoost (xgb) is widely supported and versatile, hence it was chosen predominantly for the tuning tasks. Different feature types and loss types offer flexibility depending on the tuning scenario.

### 3.4.4 Number of Trials

The number of trials is a crucial parameter that dictates how many different configurations the tuner will try. A higher number of trials increases the likelihood of finding the best configuration but also increases the tuning time. If the time budget is large, it can be set to a large amount, which will also make the tune run longer. For a production job, it is recommended to set the number of trials larger than the value of 20 used here. For CPU tuning, we recommend 1500 trials, and for GPU tuning, 3000-4000 trials. The number of trials required can depend on the particular model and processor, so it's worth spending some time evaluating performance across a range of values to find the best balance between tuning time and model optimization. Due to time constraints, I set the number of trials to 10, but this is not recommended for production use.

| **Feature Types for XGBoost Tuners** | |
|---|---|
| **itervar** | Uses features extracted from IterVar (loop variable). |
| **knob** | Uses the flattened ConfigEntity directly. |
| **curve** | Uses sampled curve features (relation features). |
| **Choosing the Feature Type** | |
| For single task tuning | 'itervar' and 'knob' are good. 'Itervar' is more accurate, but 'knob' is much faster. |
| For cross-shape tuning (e.g., many convolutions with different shapes) | 'itervar' and 'curve' have better transferability, while 'knob' is faster. |
| For cross-device or cross-operator tuning | Only 'curve' is recommended. |
| **Loss Types for XGBoost Tuners** | |
| **reg** | Uses regression loss to train the cost model, predicting normalized flops. |
| **rank** | Uses pairwise rank loss to train the cost model, predicting relative rank scores. |
| **rank-binary** | Uses pairwise rank loss with binarized labels to train the cost model, predicting relative rank scores. |

Figure 2: Tuning options for XGB tuner

### 3.4.5 Early Stopping

Early stopping is a technique used to terminate the tuning process early if no better configurations are found within a certain number of trials. This helps in saving time and computational resources. The early stopping parameter specifies the minimum number of trials to run before applying a condition that stops the search early. A larger value increases the tuning time but may lead to better configurations. The measure option indicates where the trial code will be built and where it will be run.

### 3.4.6 Log File

The log file option saves the tuning data to a specified file. This file can be used later to apply the best configurations found during the tuning process. Typically, the log file is named after the model's name with a `.log` extension. If transfer learning is enabled, the log file is temporarily named with a `.log.tmp` extension to indicate that it is still being refined.

The log file contains detailed records of each tuning trial, including:

- **Configuration Parameters**: The specific parameters used in each tuning trial.

- **Performance Metrics**: Metrics such as execution time, memory usage, and any errors encountered.
- **Best Configuration**: The configuration that provided the best performance during the tuning process.

This comprehensive logging allows for reproducibility, further analysis, and incremental improvements in future tuning sessions.

#### 3.4.7 Callbacks

Callbacks are functions that can be used to monitor and manage the tuning process in AutoTVM. Below are some common callbacks:

**Progress Bar** This callback displays a progress bar during the tuning process. The progress bar provides a visual representation of the tuning progress, showing how many trials have been completed and how many are remaining. The 'prefix' parameter allows you to add a custom prefix to the progress bar, which can be useful for identifying different tuning tasks.

The tuning process involves optimizing a deep learning model using TVM's autotuning capabilities. The output of the tuning process is detailed below, showcasing the performance improvements across different tasks.

```
[Task  1/ 6]  Current/Best:  855.43/ 855.43 GFLOPS
| Progress: (18/18) | 35.78 s Done.
[Task  2/ 6]  Current/Best: 3371.13/3883.70 GFLOPS
| Progress: (608/1000) | 1453.99 s Done.
[Task  3/ 6]  Current/Best: 3685.84/3896.92 GFLOPS
| Progress: (1000/1000) | 2424.65 s Done.
[Task  4/ 6]  Current/Best:  792.60/1681.77 GFLOPS
| Progress: (800/1000) | 1217.88 s Done.
[Task  5/ 6]  Current/Best:  160.49/1057.17 GFLOPS
| Progress: (816/1000) | 1230.03 s Done.
[Task  6/ 6]  Current/Best: 2586.92/3573.88 GFLOPS
| Progress: (608/1000) | 1176.10 s Done.
```

**Explanation of the Autotuning Process:**

- **Objective:** The goal of autotuning in TVM is to find the optimal set of configurations (such as loop unrolling factors, tiling sizes, etc.) that maximize the performance of a deep learning model on a specific hardware target (like GPU or CPU).
- **Tasks:** Each task corresponds to a specific configuration being evaluated during the autotuning process. These configurations are typically generated based on heuristics and historical performance data.
- **Current/Best GFLOPS:** This metric measures the floating-point operations per second (GFLOPS) achieved by the current configuration and the best configuration found so far for each task.
- **Progress:** Indicates the number of iterations completed out of the total iterations scheduled for each task. More iterations allow the autotuner to explore a broader range of configurations.
- **Time:** The elapsed time for each task, showing how long it took to evaluate the configurations up to that point.

**Conclusion:** The autotuning process involves iterative evaluation of various configurations to determine the optimal settings for maximizing performance. The output provides insights into the progress made across multiple tasks, each representing a different set of configurations being tested. This process is crucial for adapting deep learning models efficiently to different hardware architectures and achieving high-performance execution.

**Log to File** This callback logs the tuning records into a specified file. Each row in the log file is stored in the format of 'autotvm.record.encode'. The log file contains detailed information about each tuning trial, including the configuration and performance results. The 'tmp_log_file' parameter specifies the filename where the logs will be saved. This is useful for later analysis or for resuming tuning from where it left off.

**Other Callbacks** Additional callbacks can be implemented to extend the functionality of the tuning process. These might include custom logging, early stopping criteria based on performance thresholds, or integration with external monitoring tools. Callbacks provide a flexible mechanism to control and monitor the tuning process according to specific requirements. While many callbacks exist to enhance the tuning process, the ones I have primarily used are 'progress bar' and 'log to file'.

#### 3.4.8 Investigation and Debugging Process

To enhance the tuning process within Apache TVM, an investigation into the behavior of *measure.py* and *measure_methods.py* was conducted [**?**]. Key steps included:

- Identification of TVM's storage of temporary files in `/tmp`, necessitating the clearing of approximately 9GB of files to facilitate re-execution of extraction tasks for detailed examination.

- Modification of *measure_methods.py* to facilitate debugging:
  - Prevented immediate deletion of temporary files by setting deletion flags to `False` (Lines 100 and 103).
  - Introduced print statements at critical points (Lines 131 and 137) to capture build information and input data for thorough analysis.
- Implementation of these modifications within a duplicate directory `tvm-hack` and a corresponding `conda` environment.
- Despite initial modifications, challenges persisted, including compilation errors and measurement timeouts during task execution.
- Creation of a dedicated Jupyter notebook to manually instantiate the builder and runner, enabling detailed inspection of paths, arguments, and local execution outcomes.

This process provided deeper insights into the internal workings of TVM's tuning mechanisms, facilitating subsequent optimizations and enhancements.

### 3.4.9 Output Binaries

We managed to extract various information such as the temporary directory where the builder saves files and the attributes and arguments used in the build process. For instance:

```
BuildResult(filename='/tmp/tmp3_vzuu7b/
tmp_func_80854941b11475be.tar',
arg_info=(((10, 64, 112, 112), 'float32'),
((64, 3, 7, 7), 'float32'),
((10, 3, 224, 224), 'float32')),
error=None, time_cost=1.2738652229309082)
```

The filename is a `.tar` file that, when untarred, produces two object files: `devc.o` and `lib0.o`.

### 3.4.10 Challenges and Future Steps

Despite the progress, challenges such as persistent measurement timeouts and compilation errors remain. Future steps include:

- Further investigation into the causes of measurement timeouts and compilation errors.
- Refining the manual instantiation process of the builder and runner to gain deeper insights.
- Enhancing the debugging process by logging more detailed information and exploring alternative configurations and targets.

### 3.4.11 Other Requirements for Autotuning

In the simplest form, tuning requires you to provide three things:

- The target specification of the device you intend to run this model on.
- The path to an output file in which the tuning records will be stored.
- A path to the model to be tuned.

Additionally, for effective tuning, you should consider:

- Properly setting the target and device parameters in TVM to match the actual deployment environment.
- Using realistic input data during tuning to ensure the performance improvements are valid for real-world use cases.
- Reviewing the tuning logs to understand which configurations perform well and why, which can provide insights for further optimization.

By following these guidelines and adjusting the tuning parameters appropriately, you can significantly improve the runtime performance of your models on TVM.

## 3.5 Compiling Complex Models in TVM

After successfully autotuning simple models for image classification, such as ResNet, there was an idea to employ more complex models that could benefit from using many neural network operators (not only convolutional layers as in ResNet). In this way, I started experimenting with Visual Transformers (ViT) for image classification. Unfortunately, due to the complications mentioned in the previous section, they couldn't be run. After that, I began tuning BERT for natural language processing tasks. Here, because the tune extraction isn't immediately supported by the model, it is extracted with the "extract from program" method [13].

## 4 Database Implementation Application

In the realm of machine learning and deep learning model optimization, TVM's AutoTune offers a robust technique for enhancing model performance on specific hardware platforms. However, the process can be intensive and inaccessible to many due to its complexity and time requirements. To address these challenges, I propose a streamlined approach: consolidating optimized models into a dedicated database.

The rationale behind this approach is to eliminate the recurrent need for exhaustive tuning efforts, making pre-tuned models readily accessible to users. Unlike TVM's `log_to_database` approach, which records tuning configurations directly into a database, our method focuses on storing the optimized Relay modules of models. This allows for immediate deployment and execution without the overhead of recompiling libraries, thereby significantly reducing runtime and resource consumption.

### 4.1 Programs

To achieve this, I developed two primary components: **AutoTuning** and **RunningTVM**.

- **AutoTuning:** This component automates the model tuning process and stores the optimized configurations. Initially, I saved the tuning logs, but later transitioned to serializing the compiled Relay modules. This approach, as detailed in TVM's module serialization documentation [7], involves saving the Relay library as a shared object (.so) file, which encapsulates the optimized model. The implementation evolved from an interactive Jupyter notebook (`.ipynb`) to a command-line executable Python script (`.py`), allowing for parameterized execution:

  ```
  python3 AutoTuning.py cuda ResNet-18 100
  ```

  This command automates the tuning of ResNet-18 on a CUDA GPU with a batch size of 100, streamlining the process for users.

- **RunningTVM:** This Jupyter notebook (`.ipynb`) component facilitates the deployment of tuned models. Users can specify the model, network architecture, and batch size to load and execute the optimized Relay module. It incorporates sample inputs from ImageNet, offering a seamless environment for testing and deploying models post-tuning.

### 4.2 SQL Database

Centralizing optimized models and their configurations in an SQLite database provides significant benefits. It enables efficient access and retrieval of models based on specific criteria such as the device framework type (e.g. CUDA for NVIDIA GPU), network architecture (e.g., ResNet), and batch size. This structured storage not only simplifies model deployment but also improves reproducibility and scalability across diverse computing environments. Table 1 presents a summary of the total tuning sessions and configurations conducted on the corresponding machine for this database implementation.

| id | framework | network | batch size |
|---|---|---|---|
| 1 | cuda | resnet-18 | 10 |
| 2 | cuda | vgg-11 | 2 |
| 3 | cuda | mobilenet | 100 |
| 4 | cuda | inception_v3 | 50 |
| 5 | cuda | squeezenet_v1.1 | 1 |
| 6 | cuda | resnet-18 | 1 |
| 7 | cuda | resnet-18 | 10 |
| 8 | cuda | resnet-18 | 100 |
| 9 | cuda | resnet-18 | 200 |
| 10 | cuda | resnet-18 | 256 |
| 11 | cuda | resnet-152 | 1 |
| 12 | cuda | vit | 1 |
| 13 | cuda | vit | 10 |
| 14 | cuda | vit | 256 |
| 15 | llvm | resnet-18 | 1 |
| 16 | llvm | resnet-18 | 10 |
| 17 | llvm | resnet-18 | 25 |
| 18 | llvm | resnet-18 | 50 |
| 19 | cuda | bert | 1 |
| 20 | cuda | bert | 5 |
| 21 | cuda | bert | 10 |
| 22 | cuda | bert | 15 |

Table 1: Experiment Configurations satored inside the SQL Database

### 4.3 Dataset

The implementation leverages a straightforward file path schema within the database, ensuring transparency and flexibility in storing model configurations:

```
filepath="path/to/database/{framework}/{network}
/{batch_size}"
```

### 4.4 Summary

In summary, the implementation of this database marks a significant advancement in the usability and accessibility of TVM for machine learning tasks. By pre-tuning models and storing the optimized configurations in a centralized database, I have created a resource that can significantly reduce the setup time for new projects and streamline the deployment process. This database allows users to quickly access and deploy highly optimized models without the need for extensive retuning, which can be both time-consuming and computationally expensive.

Moreover, the use of SQLite for tracking the tuned settings ensures that all configurations are systematically organized and easily retrievable. This structure supports scalability, as new models and configurations can be added to the database over time, further enriching the resource pool.

The two primary components of this implementation, **AutoTuning** and **RunningTVM**, provide a robust framework for both tuning and deploying models. The **AutoTuning** script automates the tuning process and ensures that the resulting configurations are saved in a standardized format. The **RunningTVM** notebook then allows users to seamlessly load and run these tuned models with specified parameters, facilitating a smooth and efficient workflow.

In a corporate setting, this database could serve as a crucial tool for deploying machine learning models at scale. By leveraging pre-tuned models, companies can offer faster and more reliable services to their clients. The database can be integrated into the company's existing infrastructure, enabling automated updates and continuous improvement of model performance. Overall, this project lays the groundwork for a more efficient and scalable approach to machine learning model deployment.

## 5 Evaluation

**System Specifications** The experiments were conducted on a system equipped with an NVIDIA RTX 2080 Ti GPU. The system initially ran CentOS but was later upgraded to Ubuntu Linux for better compatibility and support with the required software components. The versions of the software components used in the system were carefully chosen to ensure seamless integration and optimal performance. The final versions were determined after thorough testing and validation:

- **Operating System**: Ubuntu Linux
- **GPU**: NVIDIA RTX 2080 Ti
- **TensorFlow**: 2.13
- **CUDA**: 11.8
- **CuDNN**: 8.6
- **Python**: 3.11

These versions were selected based on their compatibility with each other and the TVM framework. The initial setup process involved identifying and updating to these specific versions to ensure that the system components worked together seamlessly.

To optimize performance for the NVIDIA 2080 Ti GPU, I had to specify the correct compute capability (SM version) for the GPU. The necessary flag to set the appropriate SM version was:

```
-sm=sm_75
```

This flag ensures that the CUDA code is compiled for the correct architecture specific to the 2080 Ti GPU [**?**].

**Timing Evaluation** Throughout this thesis, a specific timing algorithm was employed to measure the performance of the proposed method. The timing process consisted of the following steps:

1. **Warm-up Phase:** The machine was primed by executing the algorithm multiple times to stabilize any initial performance variations.

2. **Measurement Phase:** The algorithm was executed 10 times consecutively to collect timing data.

The collected timing data was analyzed using statistical measures to provide a comprehensive evaluation of performance. The mean, median, and standard deviation (std) of the execution times were calculated from the collected data. Among these measures, the mean time was specifically chosen to represent the expected timing, as it provides a balanced representation of the algorithm's performance under typical conditions.

To ensure reliability and accuracy, all timing experiments were conducted under consistent computational conditions, including processor load and environmental factors.

**Research Questions** The following sections will address the technical questions that arose during the course of this research. These questions were critical to understanding the capabilities and performance of the TVM framework:

- **Question 1:** Where are the TVM files saved?
- **Question 2:** In tuning, which tuner should be used for faster tuning and which for achieving better results?
- **Question 3:** Is TVM, along with AutoTVM, faster than other frameworks (e.g., PyTorch) on CPU and GPU?
- **Question 4:** What are the differences between autotuning for specific neural networks and autotuning for all neural networks?
- **Question 5:** How does TVM demonstrate its effectiveness in complex model architectures?

## 5.1 TVM File Management

During the course of experimentation, significant challenges arose concerning the management of TVM's temporary files, which are stored by default in `/tmp`. In order to gain clearer insights into the functionality and impact of these files, approximately 9GB of accumulated temporary data on the machine were systematically cleared. This process was crucial for enabling the re-execution of TVM's task extraction (`tvm-extract-tasks`), thereby allowing for a deeper understanding of each file's role.

Upon analysis, the process revealed the following workflow: the builder generated a `tvm_tmpzz4_s6p/kernels.cu` file, which subsequently compiled into an executable CUDA file. This file was then packaged into a tar archive located within a randomly generated 64-bit number directory (e.g., `/tmp/tmpuy0f0_5r/tmp_func_dba47db8cbef2ce0.tar`), containing compiled objects (`devc.o` and `lib0.o`). These archives were passed to the runner for execution. [1]

Further investigation indicated that each tuning task and configuration space initiated the creation of a new tar archive. This systematic approach ensured that each task and configuration were appropriately encapsulated for execution within TVM's framework.

## 5.2 Evaluation of Autotuning Optimizers

When attempting to tune with a large batch size or for an extended duration, basic tuning options often fail, resulting in broken pipe errors and suboptimal tuning results. To address this issue, various tuner options were tested on a batch size of 100.

**Tuner with XGBoost**

- **Parameters**: earlystopping=50, tuner=xgb
- **Observations**: Many trials encountered warnings about invalid schedules and some resulted in broken pipes. Performance was decent but did not surpass PyTorch.

**Knob Tuner with XGBoost**

- **Parameters**: earlystopping=50, tuner=xgb
- **Observations**: Similar issues with invalid schedules and broken pipes. Performance was better than untuned TVM but worse than PyTorch.

**Curve Tuner with XGBoost Curve**

- **Parameters**: earlystopping=50, tuner=xgb_curve
- **Observations**: Several tasks failed to find valid tunes, but managed to achieve a good tune in successful cases.

**IterVar Tuner with XGBoost IterVar**

- **Parameters**: earlystopping = 50, tuner = xgb_itervar
- **Observations**: Most tasks could not provide a valid tune, indicating limited success.

**Rank Tuner with XGBoost**

- **Parameters**: earlystopping = 50, tuner = xgb_rank
- **Observations**: Performance was not satisfactory compared to other options.

**Genetic Algorithm (GA) Tuner**

- **Parameters**: earlystopping = 50, tuner = ga_tuning
- **Observations**: Tuning progress was slow with many errors. While tuning was faster, results were significantly worse.

**GA Tuner with Extended Early Stopping**

- **Parameters**: earlystopping = 600, tuner = ga_tuning
- **Observations**: Despite warnings and errors, performance was the best observed so far.

**Tune Tuner with Extended Early Stopping**

- **Parameters**: earlystopping = 600, tuner = xgb
- **Observations**: Runner settings were adjusted for longer runs. Results were promising, with improved performance metrics.

Based on the conducted tests, the performance of different tuners varies depending on the tuning duration and the complexity of the task.

- **GA Tuner**: The GA tuner proved to be the fastest and most effective for short and fast tuning sessions. It excelled in quickly finding decent configurations. However, its performance degraded in longer tuning sessions.
- **XGBoost (Rank, IterVar)**: For comprehensive and longer tuning sessions, the XGBoost tuners, specifically Rank and IterVar, consistently produced the best results. These tuners showed robust performance across a wide range of tasks and tuning durations.
- **Conclusion**: Choose the GA tuner for quick, short tuning sessions where speed is critical. For more thorough and extended tuning sessions, opt for XGBoost tuners such as Rank or IterVar to achieve the best overall results.

## 5.3 Comparison Methodology

To demonstrate the effectiveness of TVM, I compared a deep learning model's performance using a standard compiler backend like PyTorch against TVM. The comparison involved several steps:

### 5.3.1 GPU (CUDA)

First, I conducted a basic image classification task using the ResNet-18 model and the ImageNet dataset on GPU using CUDA with PyTorch. The model was configured to utilize ImageNet's pretrained weights, and I successfully performed image classification. Subsequently, I measured the classification times using a timing algorithm.

Next, I implemented the inference model in TVM for GPU. Initially, this model ran without any autotuning. I used the same ResNet-18 model, processed it through Relay to create a TVM module, and obtained classification results while measuring the runtime.

The final step involved applying autotuning, as detailed in the previous section on Methodology and Autotuning Options. The autotuning process encompassed multiple tuning sessions, some lasting several days. Upon achieving successful image classification with the autotuned model, I conducted runtime measurements again.

The processes depicted in Figure 3 illustrate significant insights into the performance comparison between PyTorch and TVM on GPU. Initially, TVM without autotuning shows competitive performance with PyTorch, particularly noticeable at batch size 1. However, as batch sizes increase, PyTorch maintains more stable performance while TVM without autotuning exhibits slower execution times.

Remarkably, after applying autotuning (AutoTVM), TVM achieves improved performance across all batch sizes tested, surpassing both PyTorch and its initial performance without autotuning. This enhancement underscores the effectiveness of TVM's autotuning capabilities in optimizing inference performance, often outperforming hand-tuned implementations in traditional frameworks like PyTorch.

For Figure 3 the highest achievable batch size was limited slightly above 256 due to GPU memory constraints.

This comprehensive evaluation highlights the potential of TVM as a competitive alternative for optimizing deep learning inference on GPUs, demonstrating substantial performance gains through automated tuning processes.

### 5.3.2 CPU (LLVM)

Similarly, I conducted experiments on CPU using LLVM as the backend compiler. The ResNet-18 model with ImageNet was used for image classification, and the timing measurements were taken for batch sizes of 1, 10, 25, and 50. These measurements were compared and visualized using plot graphs.

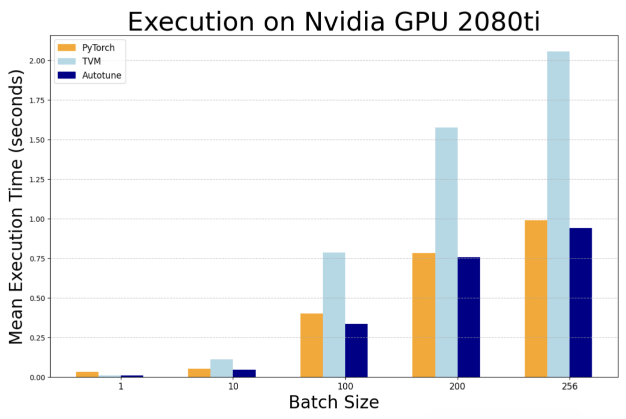

Figure 3: Mean timing results comparing the PyTorch framework, TVM without autotuning, and TVM with complete autotuning (AutoTVM) using CUDA on NVIDIAs RTX2080ti Graphics Card. Batch sizes tested include 1, 10, 100, 200, and 256.

Figure 4 illustrates the mean timing results for the PyTorch framework, TVM without autotuning, and TVM with complete autotuning (AutoTVM) on CPU using LLVM across different batch sizes. While TVM did not surpass PyTorch in performance, it showed significant improvement over its own baseline without autotuning. The time constraints limited the opportunity for more detailed and potentially longer tuning sessions, which might have closed the performance gap with PyTorch. Despite this, the observed improvements with minimal manual effort demonstrate the potential and effectiveness of TVM's autotuning capabilities.

## 5.4 Comparison of Neural Network Operators

When optimizing computational graphs in TVM, it is possible to compile a model focusing on specific neural network operators (NNOps), such as `conv2d`. This section discusses the differences and when to use this approach.

### 5.4.1 ResNet-18

For a model like ResNet-18, which predominantly uses convolution operations, tuning exclusively with `conv2d` makes sense. By restricting the optimization process to `conv2d` operators, the autotuning efforts are focused on the most computationally intensive parts of the model. Tuning with all neural network operators (NN Ops) would only complicate the process without significant benefits. This is the primary rationale for specifying convolution operations.

Figure 5 presents a comparison between two autotuning runs on CUDA: one specifying only `conv2d` for ResNet-18 and the other allowing any neural network operator to be tuned. The experiments show no significant time difference, as the resulting mean times are almost identical.

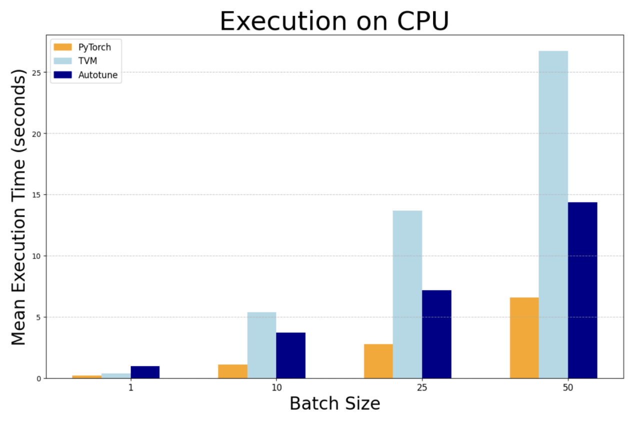

Figure 4: Mean timing results comparing the PyTorch framework, TVM without autotuning, and TVM with complete autotuning (AutoTVM) for CPU using LLVM. Batch sizes tested include 1, 10, 25, and 50.

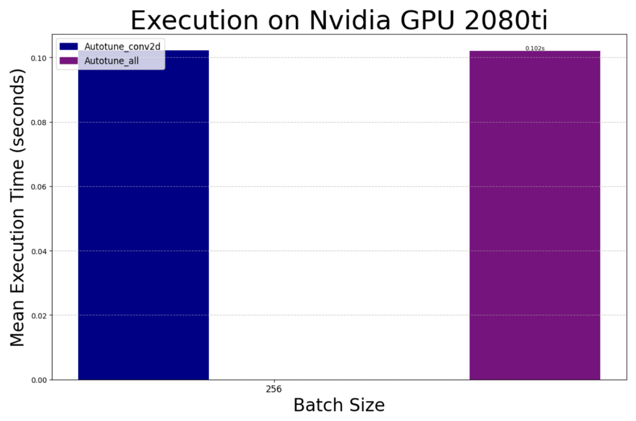

Figure 5: Comparison of two AutoTVM results for ResNet-18: one specifying only `conv2d` operators and the other allowing any neural network operator. Both runs utilized the GA tuner with early stopping set to 120, running on GPU with CUDA.

Reviewing the log files from each run reveals the difference in the operators tuned:

- **`conv2d`-only run:** This run focused exclusively on tuning convolution operations, specifically `conv2d` operators.

- **All operators run:** This run included a variety of neural network operators such as:

  - `dense_large_batch.gpu`
  - `dense_small_batch.gpu`
  - `conv2d_nchw_winograd.cuda`
  - `conv2d_nchw.cuda` (multiple instances)

The detailed log entries for the all-operators run show a mix of dense and convolution operators, while the `conv2d`-only run consistently tuned `conv2d_nchw.cuda` and `conv2d_nchw_winograd.cuda` operators. By avoiding the tuning of less significant operators, the autotuning process becomes more efficient, leading to faster tuning times but without any performance gains.

In conclusion, focusing the optimization process on specific neural network operators, such as `conv2d`, accelerates the tuning process for deep learning models but does not necessarily enhance their performance. Since the ResNet-18 model predominantly uses convolution operations, directing the tuning to these operators allows for faster completion of the tuning process. However, this approach does not result in superior execution efficiency compared to tuning all operators, as the model's performance is already optimized for convolutions.

#### 5.4.2 Vision Transformer (ViT)

Continuing the comparison across all neural network operators versus focusing on `conv2d`, I selected the Vision Transformer (ViT) model, which incorporates a diverse range of operations. The decision to tune the ViT model with a comprehensive autotuning approach (including all neural network operators) versus exclusively using `conv2d` operators yielded distinct tuning results, highlighting the impact of comprehensive optimization.

When tuning the ViT model exclusively with `conv2d`, the autotuning process focused on optimizing `conv2d_nchw.cuda` operations, generating four instances of this operator. In contrast, using all neural network operators during tuning resulted in a broader spectrum of optimized operations. Specifically, the log files from the all-operators tuning session revealed the following sequence of operations:

- `dense_large_batch.gpu`
- `dense_small_batch.gpu`
- `conv2d_nchw.cuda`

An attempt to run the ViT model on an NVIDIA RTX 2080 Ti GPU encountered a limitation related to shared memory per block. The specific error encountered was:

```
CUDA ptxas Error: "function uses too
much shared data"
```

This issue underscores the importance of considering hardware constraints when optimizing and tuning models. For complex models like ViT, which may exceed the memory architecture limits of the target device, careful consideration of model complexity and hardware capabilities is essential [11].

## 5.5 BERT Model

In the pursuit of a sufficiently complex model to optimize using AutoTVM, I explored the Bidirectional Encoder Representations from Transformers (BERT) model. Following a tutorial on bridging PyTorch and TVM with this model [14], BERT's intricate architecture served as a proof of concept that models can be seamlessly integrated into TVM and achieve efficient execution. This also underscored TVM's versatility in handling models beyond traditional image classification.

A primary challenge encountered was the distinction in batching for language models compared to images. In my experiments, a batch was defined as the product of the batch size and the token and segment tensors used as inputs. Furthermore, due to limitations in GPU memory, the autotuning process was constrained to batch sizes under 20.

The tuning log for the BERT model revealed the following optimized operations:

- `dense_small_batch.gpu` (repeated 6 times)
- `batch_matmul.cuda` (repeated 30 times)

Figure 6 depicts the runtime results for the BERT model on an RTX 2080 Ti with batch sizes of 1, 5, 10, and 15, comparing PyTorch, Inference TVM, and AutoTuned TVM runs. While the autotuning did not consistently outperform PyTorch across all batch sizes, longer tuning durations hold promise for achieving superior performance. Of particular note is the comparison between AutoTVM and inference TVM, highlighting AutoTVM's capability in optimizing the model's execution efficiency.

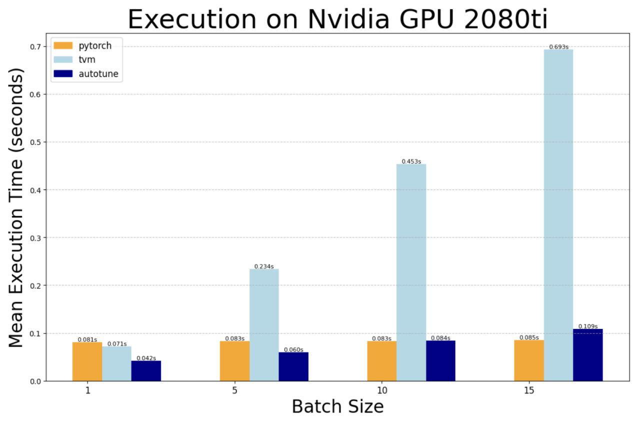

Figure 6: Runtime results for the BERT model on an RTX 2080 Ti with batch sizes of 1, 5, 10, and 15. The input text used was "[CLS] Who was Jim Henson? [SEP] Jim Henson was a puppeteer [SEP]".

In conclusion, the autotuning results for the BERT model underscore TVM's capability to handle complex NLP models alongside traditional image processing tasks. This versatility positions TVM as a robust tool for a wide array of deep learning applications, demonstrating its potential for optimizing performance across diverse model architectures.

## 6 Conclusion

This work addressed and answered several key research questions related to the use of TVM as a machine learning compiler:

- **How does TVM compare with other machine learning compilers?**
- **How can we overcome the time-consuming AutoTVM process?**

TVM, as an end-to-end compiler stack for deep learning, provides a comprehensive set of optimizations for a wide variety of hardware backends. Unlike most deep learning (DL) compilers, which tend to be graph-based, TVM's Relay Intermediate Representation (IR) is a functional language that supports common programming constructs such as conditionals and loops. This functional nature gives TVM better support for more complex, dynamic models. TVM generates its own low-level, hardware-specific code for a diverse set of backends and utilizes an ML-based cost model called AutoTVM, which adapts and improves code generation based on continuous data collection [2].

Moreover, TVM integrates established optimization techniques in a novel manner, setting it apart from other ML compilers. This versatility allows TVM to be adaptable to various hardware architectures, making it a revolutionary tool in the field of deep learning.

During this thesis, it was demonstrated that TVM for GPU can outperform PyTorch using CUDA, with the significant advantage that AutoTVM tuning requires little to no manual effort. Although this was proven for an NVIDIA GPU, the groundwork laid here can be extended to any hardware, especially custom hardware. Development teams can utilize TVM to optimize performance without the significant monetary and effort costs typically associated with manual tuning.

The autotuning process of TVM is indeed time-consuming, often taking days to complete for complex models and hardware configurations. To address this challenge, a database application was developed to save each tuning result. This solution ensures that once a tune is completed, it can be stored and reused, significantly reducing the time required for future optimizations on the same or similar hardware. This approach can be highly beneficial for companies, enabling them to have hardware ready to run optimized models efficiently due to the saved tuning results.

Overall, this thesis underscores TVM's potential as a versatile and powerful compiler for deep learning, capable of delivering significant performance improvements with minimal manual intervention. The developed solutions for autotuning and tuning data management further enhance TVM's usability, paving the way for its broader adoption in both academic and industrial settings.

## 7 Acknowledgments

Words cannot express my gratitude to my professor and guide, Mr. Christos Kozanitis, for his invaluable patience, guidance, and unwavering support throughout this research journey. His insightful feedback and encouragement have been instrumental in shaping this thesis.

I am also deeply grateful to Mr. Aggelos Mpilas and Mr. Polivios Pratikakis for their generous sharing of knowledge and expertise. Their mentorship and willingness to discuss ideas have enriched my understanding of Computer Architecture and VLSI Systems (CARV) significantly.

Additionally, I extend heartfelt thanks to Klodjan Hidri and Stelios Mauridis for their unwavering technical support and assistance throughout the research process. Their expertise and prompt assistance in overcoming technical challenges were indispensable.

This research would not have been possible without the support and encouragement of these individuals, and I am truly grateful for their contributions to my academic and professional development.

## References

[1] deep-learning-compilers. https://github.com/apache/tvm/tree/main/python/tvm/autotvm/measure. Accessed: July 7, 2024.

[2] deep-learning-compilers. https://unify.ai/blog/deep-learning-compilers#tvm. Accessed: July 7, 2024.

[3] Introduction to boosted trees. https://xgboost.readthedocs.io/en/stable/tutorials/model.html, 2024. Accessed: July 4, 2024.

[4] AMKRISHNA2910. Onnx resnet-50 model file. https://github.com/onnx/models/blob/main/validated/vision/classification/resnet/model/resnet50-v2-7.onnx. *onnx resnet 50 model file*.

[5] CHEN, T., MOREAU, T., JIANG, Z., ZHENG, L., YAN, E., SHEN, H., COWAN, M., WANG, L., HU, Y., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. TVM: An automated end-to-end optimizing compiler for deep learning. https://www.usenix.org/conference/osdi18/presentation/chen, Oct. 2018. Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

[6] CHEN, T., ZHENG, L., YAN, E., JIANG, Z., MOREAU, T., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. Learning to optimize tensor programs. http://papers.neurips.cc/paper/7599-learning-to-optimize-tensor-programs.pdf, 2018. Accessed: July 7, 2024.

[7] CONTRIBUTORS, A. T. Introduction to module serialization. https://tvm.apache.org/docs/arch/introduction_to_module_serialization.html, 2024. Accessed: July 6, 2024.

[8] CONTRIBUTORS, A. T. tvm/python/tvm/topi/nn/__init__.py at main · apache/tvm. https://github.com/apache/tvm/blob/main/python/tvm/topi/nn/__init__.py, 2024. Accessed: 2024-07-06.

[9] FOUNDATION, A. S. Apache tvm measure_methods.py. https://github.com/apache/tvm/blob/main/python/tvm/autotvm/measure/measure_methods.py#L561, Year. Accessed: Date.

[10] JAIN, A. Data layout format. https://tvm.apache.org/docs/arch/convert_layout.html. Accessed: July 7, 2024.

[11] OVERFLOW, S. Cuda ptxas error: "function uses too much shared data". https://stackoverflow.com/questions/23648525/cuda-ptxas-error-function-uses-too-much-shared-data. Accessed: 2024-07-07.

[12] USER, S. E. Find input shape from onnx file. https://stackoverflow.com/questions/56734576/find-input-shape-from-onnx-file, 2019.

[13] VIEHMANN, T. Bridging pytorch and tvm. https://tvm.apache.org/2020/07/14/bert-pytorch-tvm, July 2020. Accessed: 2024-07-07.

[14] VIEHMANN, T. Bridging pytorch and tvm. https://tvm.apache.org/2020/07/14/bert-pytorch-tvm, July 2020. Accessed: 2024-07-07.