



저작자표시-비영리-동일조건변경허락 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.
- 이차적 저작물을 작성할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



동일조건변경허락. 귀하가 이 저작물을 개작, 변형 또는 가공했을 경우에는, 이 저작물과 동일한 이용허락조건하에서만 배포할 수 있습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

M.S. THESIS

Neural Network Translation for Flexible
Execution on SKT's AIX Accelerator

SKT AIX 가속기의 유연한 실행을 위한 신경망 해석

AUGUST 2020

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Richard Heithorst

Neural Network Translation for Flexible Execution on
SKT's AIX Accelerator

SKT AIX 가속기의 유연한 실행을 위한 신경망 해석

지도교수 Bernhard Egger

이 논문을 공학석사 학위논문으로 제출함

2020 년 06 월

서울대학교 대학원

컴퓨터 공학부

리 차 드

리차드의 공학석사 학위논문을 인준함

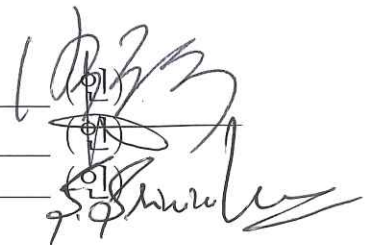
2020 년 06 월

위 원 장
부위원장
위 원

박근수

Bernhard Egger

Srinivasa Rao Satti



Abstract

Computer vision problems are best addressed by convolutional neural networks (CNNs). These resource intensive algorithms require the use of specialised accelerators when the application is particularly time critical or when power efficiency plays an important role. Accelerators' performance gains are reached at the cost of generality. In this thesis, our target hardware is SKT's AIX accelerator designed for the execution of darknet CNNs. The presented work enables the flexible execution of ONNX networks on AIX extending the accelerator's support to an additional framework. We will see the steps to take to map a very different graph structure into that of the accelerator and how we still achieve partial acceleration of networks containing unsupported operations. Running ONNX networks on AIX through our project yields very close results to native execution on ONNXRuntime. Indeed, we reach a 98% top-1 prediction match on a CIFAR10 sample.

Keywords: Computer Vision, CNN Acceleration, Network Translation

Student Number: 2018-25262

Contents

Abstract	i
Contents	iii
List of Figures	v
Chapter 1 Introduction	1
Chapter 2 Background	3
2.1 Convolutional Neural Networks for Computer Vision	3
2.1.1 Feature Maps	4
2.1.2 Convolutions	4
2.1.3 Batch Normalisation	6
2.1.4 Activation Functions	7
2.1.5 Pooling and Subsampling	7
2.2 Open Neural Network Exchange	8
2.3 SKT's Custom Accelerator: AIX	9
Chapter 3 Design	12
3.1 Overview	12

3.2	Operator Support and Graph Division	13
Chapter 4	Implementation	15
4.1	Project Architecture	15
4.2	Graph Division: Ready Queue Exploration	17
4.3	Object Oriented Approach for Common Subgraph Interface	19
4.4	Handling the Data Flow	22
4.5	Working with a Custom Tool under Development	22
Chapter 5	Test Setup	23
5.1	CIFAR10	23
5.2	Prototyping and Converting Test Networks	24
Chapter 6	Results	26
Chapter 7	Conclusion	29
	Bibliography	30
	요약	34

List of Figures

Figure 2.1	Visual representation of CNN vocabulary	5
Figure 2.2	Visual example of a convolution layer’s computation. . .	6
Figure 2.3	Activation and pooling examples	8
Figure 2.4	A simple ONNX example visualised with Netron	9
Figure 2.5	High level architecture of the AIX accelerator.	10
Figure 3.1	Graph Mapping	13
Figure 3.2	Graph Division	14
Figure 4.1	Overview of the project’s architecture.	16
Figure 4.2	Visualisation of ONNX’s network structure.	18
Figure 5.1	CIFAR-10 sample.	24
Figure 5.2	Test network.	25
Figure 6.1	Effects of calibration on the translation’s match rate. Green means AIX and ONNX classified an image iden- tically. Red indicates a mismatch.	27
Figure 6.2	Detailed results	28

Chapter 1

Introduction

The recent improvements made around machine learning and the performance of GPUs lead to a major development of computer vision. With various applications in sectors of autonomous driving, security and health for example, the interest of businesses in computer vision is increasing. GPUs became very good at executing and training deep neural networks (DNNs) but in certain particularly time critical scenarios or when power efficiency plays a major role, they are outperformed by accelerators. However, accelerators' performance gains are reached at the cost of generality.

The work we will present in this thesis is part of a project with SKT. One of their teams is developing such a neural network accelerator referred as AIX. Targeted at computer vision, it mainly aims to accelerate CNNs.

Machine learning's gain in popularity also lead to the emergence of various different frameworks, each having their own specificities. This multitude of standards does not match well with the restrictive architecture of an accelerator. In addition to its performance, an accelerator should however support popular

frameworks to be attractive.

After introducing some related concepts, we will explain how we proceed to extend the accelerator’s support to a new framework by properly mapping the original graph into required format. We also present how will are still able to accelerate networks containing unsupported operations with our flexible execution approach. We will then show how we evaluate our project and what results it yields.

Chapter 2

Background

2.1 Convolutional Neural Networks for Computer Vision

A convolutional neural network (CNN or ConvNet) is a neural network containing at least one convolutional layer. This type of network is well suited to address computer vision problems as they take advantage of weight reuse to process the large input. CNNs are usually multistage networks and therefore also belong to the deep neural network (DNN) category.

The first major application of CNNs in computer vision surely is the recognition of numbers in ZIP Codes by LeCun et al. using a three stage CNN [12] in 1989. This structure evolved from earlier works like the neocognitron [3] applied to signal processing and the studies of the visual cortex of animals [7] [8]. CNNs have since improved and became much deeper (more layers). Networks like ResNet [6] for classification or YOLO [16] for object detection perform astonishingly well in their respective fields.

We will now go over the different operations commonly used in CNNs and introduce the related vocabulary. The following explanations are based on [13] and [4]

2.1.1 Feature Maps

In this text, we will refer to feature maps (FM) as the multidimensional data tensors between the CNN layers. A layer takes an input feature map and produces an output feature map. In computer vision, the network's input is the set of pictures to process. The output is the network's prediction.

For CNNs, feature maps are generally 4 dimensional. Let us look more closely at the network's input for a moment. An image is represented as three two dimensional arrays containing the respective RGB value for each pixel. We call each of this two dimensional arrays a channel. CNNs can process batches of multiple images at a time, adding a 4th dimension to the feature map. Note that in certain papers, the authors refer to individual channels as feature map. Feature maps are also referred to as activations.

2.1.2 Convolutions

As the name suggests, convolutional layers are the main component of CNNs. They represent the computational core of this type of networks, extracting certain features from their input independently of the position. They are a more efficient version of the usual fully connected layers thanks to shared the sharing of parameters (the same weights are used for multiple outputs) and the sparse connectivity (an output depends only on a portion of the input, not its entirely).

The following explanation can be visualised in figure 2.1. A convolution is composed of N filters. Each filter is applied to the entire input feature map and

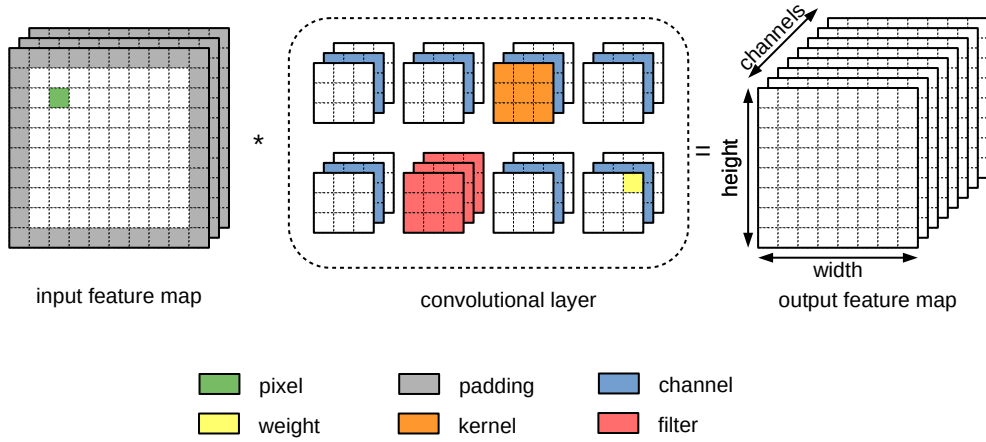


Figure 2.1: Visual representation of CNN vocabulary

yields one of the output feature map's channel. So the output feature map will have N channels. If an input feature map has C channels, each filter will have C two dimensional kernels of $H \times W$ weights. The height H and the width W are constant across the whole layer, typically 3×3 . Thus, a convolutional layer also has C channels composed of the kernels at a given index from each filter. A layer contains $N \times C \times H \times W$ weights and N biases, one for each filter.

Let i_{bchw} , respectively o_{bchw} , be the value from the input, respectively output, feature map in the b^{th} batch, the c^{th} channel, in the h^{th} row and the w^{th} column. Similarly, let w_{nchw} be a given layer's weight in the n^{th} filter, the c^{th} channel, in the h^{th} row and the w^{th} column and b_n the bias of the n^{th} filter. The formula used to compute an output pixel is the following:

$$o_{bki j} = b_k + \sum_{c=0}^{C-1} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} w_{kchw} \times i_{bc(i-H/2)(j-H/2)}$$

Zero padding is applied on the edges of the input channels to preserve the feature map dimension. The convolutional operation can be seen in figure 2.2. To

compute an output pixel, the weights of the corresponding filter are multiplied with the matching input pixels (the input pixels from all channels at the same position i, j as the computed output pixel, as well as the neighbouring ones to match the layer’s kernel size). These products are summed together and finally the filter’s bias is added.

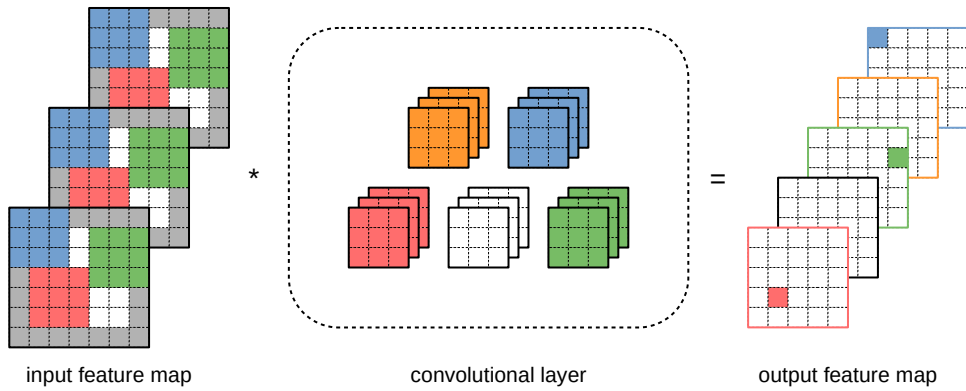


Figure 2.2: Visual example of a convolution layer’s computation. The filters’ channels match the input channels. Each filter generates one output channel by sliding over the entire input. The color code shows three examples of where the generating filter is applied to the input to compute a given output pixel.

2.1.3 Batch Normalisation

Batch normalisation [9] has been introduced in 2015 and has since received broad acceptance. The reasons behind are still debated over but the observed improvements are incontestable. Batch Normalisation consists in normalising the output of convolutional layers between batches or mini batches during training. In addition to that, a trainable mean and scale were added to it. This greatly

speeds up the training. During back propagation (training), the parameters are updated assuming that the rest of the parameters remain constant. This is not true however. All parameters are updated simultaneously. By normalising the outputs of convolutional layers, the influence of the other parameters (called internal covariate shift) is mitigated. Normalisation also makes the network more general greatly reducing overfitting. For example, the difference between two pictures with different lighting is greatly reduced through this operation.

It is now used in most convolutional networks and has opened the door to even deeper layers since their training time has now become acceptable.

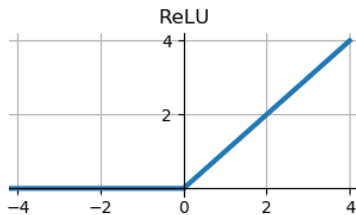
2.1.4 Activation Functions

Activation functions, also referred to as non-linear functions or layers, are applied to the outputs of convolutions. They are used to simulate the firing rate of neurons in biological brains. The sigmoid was initially used to serve that purpose. Architectures shifted to tanh instead and now the rectified linear unit (ReLU, figure 2.3a) is mostly used. Its great speedup of training by accelerating convergence and its cheap computational cost made it stick out compared to its predecessors. It simply computes $f(x) = \max(0, x)$. Alternatives like the leaky ReLU or PReLU are also commonly used to address some of ReLU's drawbacks.

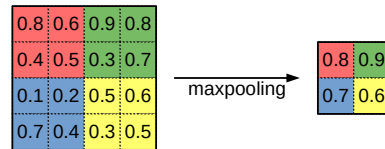
2.1.5 Pooling and Subsampling

Pooling layers are used to make the network less sensitive to small translations of the input. Maxpooling is most commonly used. It moves over the input with a given window (typically 2×2) and keeps only the maximal value of the neighbourhood. Alternatives are average pooling or L^2 . It enables to detect the presence of a certain feature in a certain area rather than an exact spot.

An aspect strongly associated with pooling is subsampling. It consists in



(a) The most common activation function, ReLU function: $f(x) = \max(0, x)$



(b) Demonstration of maxpooling with a window of 2×2 and a stride of 2.

Figure 2.3: Activation and pooling examples

sliding the pooling window with a stride bigger than one, reducing data overlapping and focusing on the most important values (figure 2.3b). This also reduces the computational complexity of the following layers as it reduces the height and width of the feature map.

The notion of subsampling is easier to understand with maxpooling but can also be applied during convolutions by increasing the stride of the filters over the input.

2.2 Open Neural Network Exchange

Open Neural Network Exchange or ONNX [2] is an open source machine learning format. It is probably easiest to keep the description of its creators:

ONNX is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers.

It enables to easily convert networks between frameworks. It is not intended

for the training or the designing of networks. ONNX instead focused on inference and flexibility. As stated, this format relies on simple operators instead of complex layers. The graph and its parameters are saved in a single file. ONNX's structure is defined in Google's protocol buffer (protobuf) [5]. ONNX graphs can easily be visualised with tools like Netron [17]. A simple example is shown in figure 2.4.

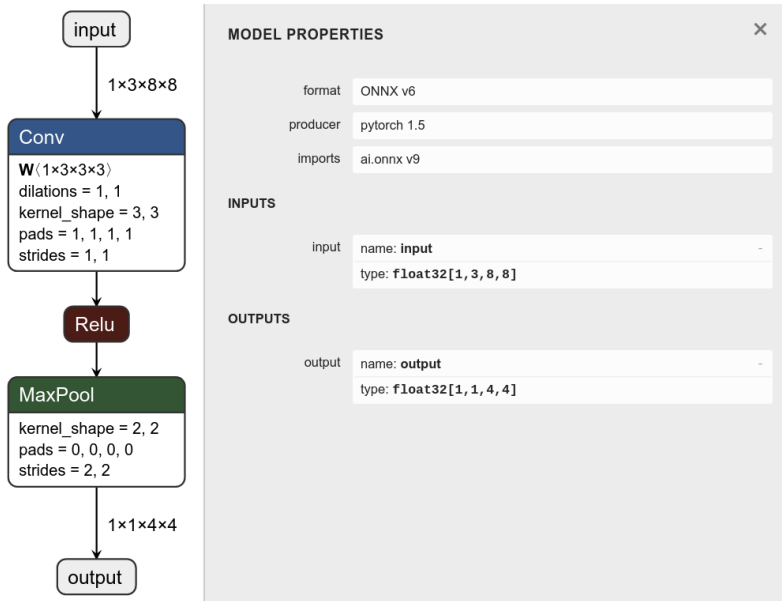


Figure 2.4: A simple ONNX example visualised with Netron [17]

2.3 SKT's Custom Accelerator: AIX

The neural network accelerator AIX [1] is developed by SKT. Its design is tightly linked to Darknet [15]. A graph structure defined in protobuf serves as high level interface. This graph is then compiled into a command graph scheduled across the accelerator's different units. AIX's architecture overview is shown in figure 2.5.

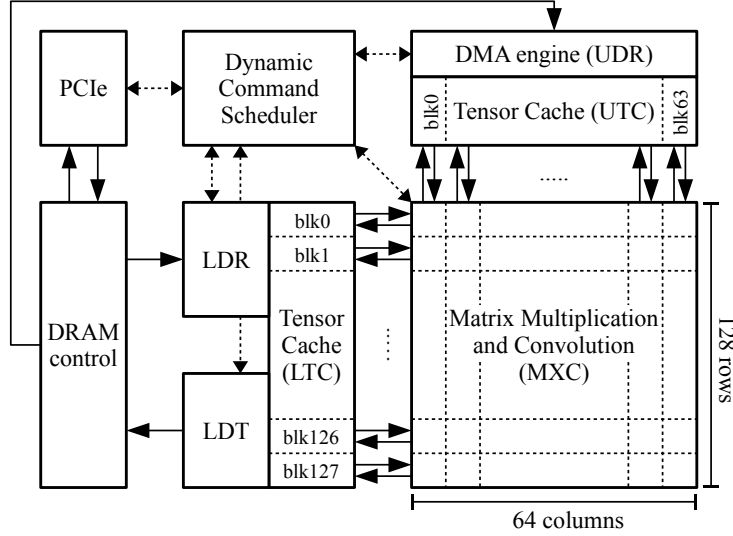


Figure 2.5: High level architecture of the AIX accelerator. The data in the tensor caches are divided into blocks that match the rows and columns of the MXC processor.

The matrix multiplication and convolution (MXC) unit is a systolic array designed to efficiently process 2D convolutions and perceptron layers and plays a key role in AIX’s acceleration abilities. Under the right conditions, it can process a convolution, a normalisation, an activation function and a pooling operation without intermediate accesses to DRAM. The dynamic command scheduler (DCS) orchestrates the other components by scheduling independent tasks in parallel as well as tracking components availability and command completion statuses. The upper data receive (UDR), left data receive (LDR) and left data transport (LDT) units are direct memory access (DMA) engines reading (receive) from or writing (transport) to the DRAM. The left tensor cache (LTC) and upper tensor cache (UTC) serve as buffers to the MXC to avoid stalling. They are flushed and refilled with data while the MXC unit processes their

previous content. The UTC holds the network's parameters, typically weights or biases, while the LTC contains tiles of the feature maps (intermediate results between layers).

Chapter 3

Design

In this section, we will look at the design of the presented work without going into the technical details (next chapter).

3.1 Overview

SKT's AIX hardware is tightly linked to darknet and only supports this framework for now. Our objective is to extend AIX to ONNX, ie. enable the execution of ONNX networks on AIX. However, as we mentioned earlier, ONNX structure relies on simple operators rather than complex all-in-one layers. This large variety of basic operators is great for versatility and multi-framework support. But when the target is a restrictive accelerator, they add a lot of complexity. In figure 3.1, we can see how nodes in the original darknet and ONNX graphs need to be merged differently to form the proper AIX corresponding nodes.

Another issue with the multitude of operators is that it is possible to get the same result with different manners. This variety also enables less standard

designs which leads us to the next part.

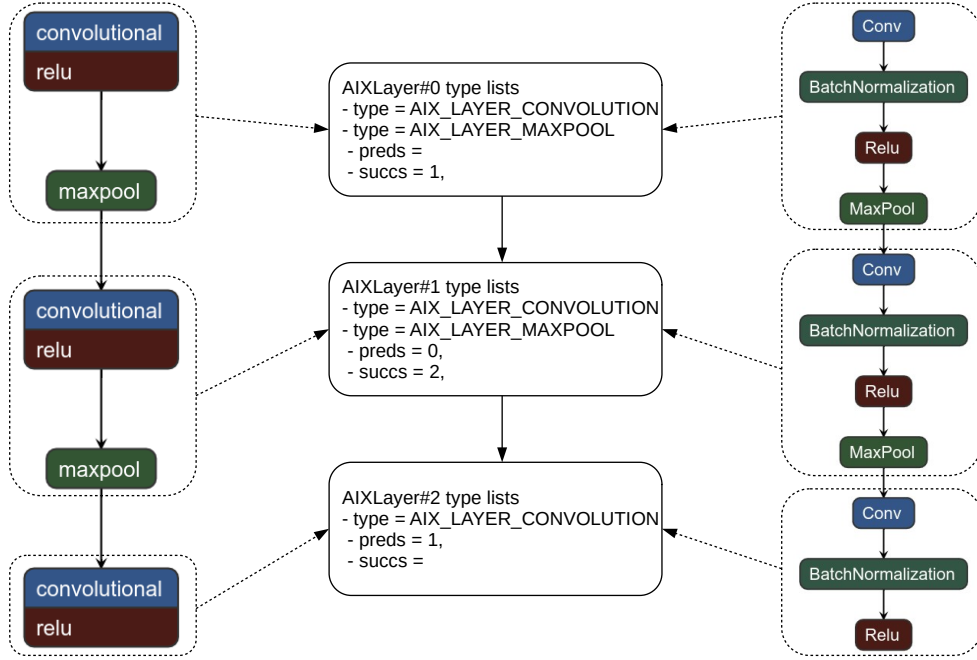


Figure 3.1: Graph mapping illustration. From Left to right, we can see representations of darknet, AIX and ONNX graphs.

3.2 Operator Support and Graph Division

AIX focuses on the acceleration of CNN's common and heavy computational parts. It has good support for convolutions, normalisation, activation layers and, in a certain extent, pooling layers. These parts are very similar across different networks. However, final layers targeted towards the classification or detection output are less uniform between networks and frameworks. Even in the company's support of darknet, final layers are executed in the native framework and not on AIX.

When executing ONNX networks on AIX, running certain parts of it in the

native framework is therefore inevitable. In addition to special final layers, some ONNX network overall contain unsupported operators. To address this issue, we need to take a flexible approach.

When translating an ONNX network, we first analyse the graph and split it into groups of consecutive nodes that run on the same backend. We then create subgraphs out of those groups to meet their respective backend requirements. Subgraphs share a common interface independently of their execution platform. We will cover the details in chapter 4. An example of graph division according to operators' support can be seen in figure 3.2.

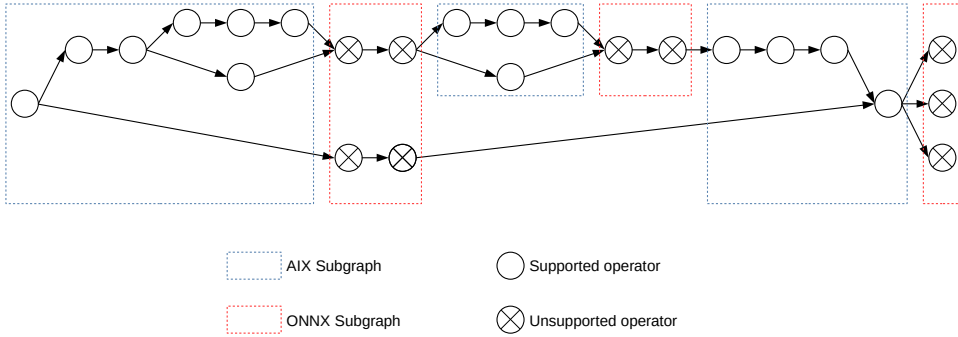


Figure 3.2: Example of graph division according to the operators' AIX support.

Once the graphs created, they can be executed independently. We handle the order of execution and manage the overall dataflow by forwarding the outputs of subgraphs as the inputs of the subgraphs that need them. Once all subgraphs are executed, we get the final output as if the network ran in one go.

Chapter 4

Implementation

4.1 Project Architecture

Figure 4.1 gives as an overview of the project’s architecture. `onnx2mxconv.py` is the main file. It contains the implementations of the `AIXRunner` class which is responsible for the initial graph analysis and splitting, the `SubGraph` classes and the `IntermediateNode` class. `onnx_utils` contains a number of helper functions that help interact with and navigate ONNX graphs. `aix_graph.cc` contains the `AIX` class that creates and runs the AIX graph given parameters from ONNX.

You notice here that the AIX API and the code that interacts with it is in written in C/C++. The rest of the project is however in Python because ONNX’s main API is in Python as well as MLPerf [14] that is a target application of the project once completed. MLPerf is an emerging benchmark to compare the performance of different machine learning hardware.

To bridge the code between languages, we made a custom Python module from the AIX C/C++ code. To achieve that, we used `pybind11` [10] and

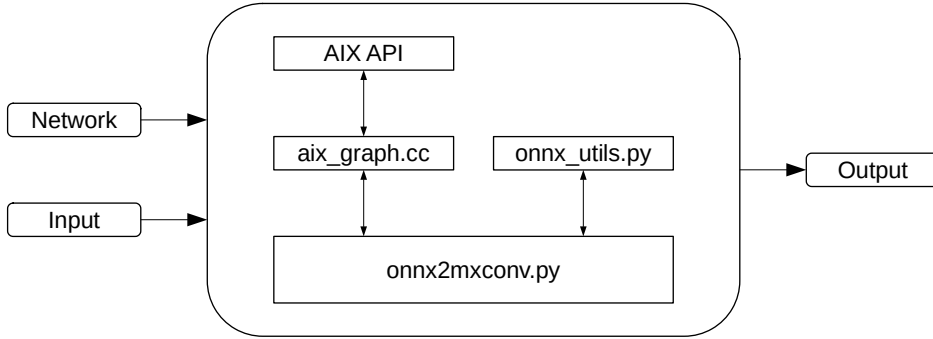


Figure 4.1: Overview of the project’s architecture.

setuptools. These tools make the link between C and Python objects and to compile the C code into a shared library. The shared library can then be imported in Python like any other module. Since the AIX API and the related simulator code is a big project, creating the module was tricky. All necessary files needed to be included properly while keeping it to the minimum. All the relative paths needed to be reflected during compilation. There were also several library compatibility issues with protobuf and ONNX. But once all these points were addressed after quite a bit of trial and error, our custom module was ready for use and the C/Python interaction worked seamlessly.

On a side note, there was one interaction between C and Python that resulted in incorrect results and took a long time to identify and fix. In the code invoking the AIXRunner we do some preprocessing on the input image using NumPy. One of those steps was to transpose the input array’s dimension to match the network’s requirements. However, when transposing dimensions in NumPy, the data in memory doesn’t change. NumPy simply changes the dimensions’ stride and therefore accesses the expected data. We then give a pointer to this data to read in C. There, the data is in its original disposition and data is

not accessed in the desired manner. Considering that there were a lot of other places the incorrect output could have come from, getting behind this took a considerable amount of time.

4.2 Graph Division: Ready Queue Exploration

As seen in the previous chapter, we need to split the original ONNX graph into subgraphs since certain operators can not be executed on AIX. Now would be a good time to talk a little about the way an ONNX graph is organised. An ONNX network is composed of metadata (like the version of the network, the used ONNX operator version etc.) and more importantly its graph. The graph itself is composed of:

- A list of nodes: each node is an operator. It is defined by its type, its name, its attributes (stride, padding, etc.). A node also contains a list of the names (strings) of its inputs and one for its output.
- A list of inputs: the classification of what is called an input is a little strange in ONNX. This list contains the graph's input (typically the input image) as well the parameter inputs for each node (weights, biases and so on) but not the intermediate activations. An input is defined by its data type and its shape.
- A list of initializers: Initializers contain the data of the networks parameters that do not depend on the input such as weights and biases. In addition to the data type and the shape, initializers contain the actual data.
- A list of outputs: As mentioned, the graph does not contain the intermediate activations' structure so this list only contains the graph's final

outputs.

Figure 4.2 shows a visualisation of this structure. Since nodes store the names of their inputs and parameters rather than their index or the actual object, accessing the graph’s different elements isn’t very straightforward. The functions in `onnx_utils.py` help address that issue. For example, they return the index of an object of a given name.

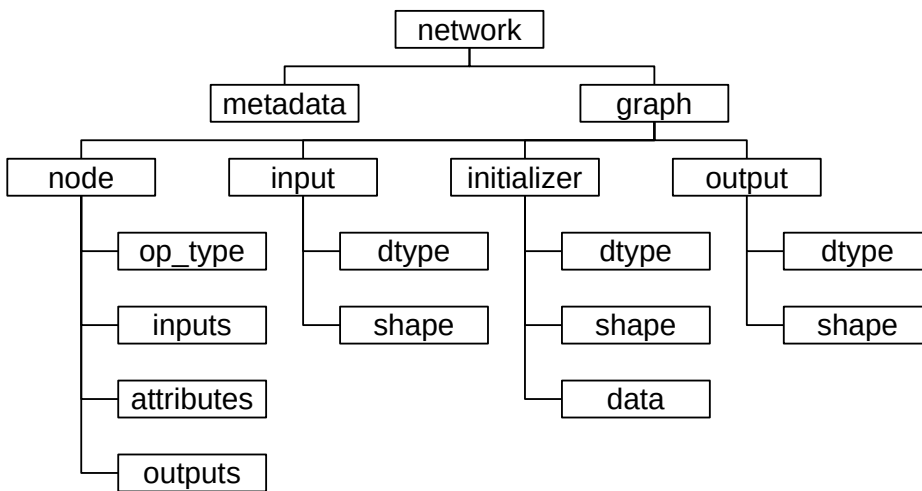


Figure 4.2: Visualisation of ONNX’s network structure.

Now that we understand the ONNX structure a bit better, let us see how split divide it. We use a form of ready queue keeping track of the backend we are currently grouping for. First of all, we add all nodes that have the graph’s input as an input to a list we will call frontier. To that frontier, we append all nodes that do not have a predecessor so that they are not forgotten during exploration. This can happen in ONNX networks when constants are fed into certain operators.

From there on, we take the first node in the frontier, and check its AIX

support. We define a static list of operator types that are always support. If the node doesn't correspond to on of those, we check we check if it matches the conditions in which certain operators can run on AIX. Otherwise, the node will run on ONNX.

If the subgraph type is not set, we set it to the backend of the current node. If the node doesn't match the current subgraph type, we move on to the next node in the frontier. For nodes that match the subgraph type, we check if all its predecessors have been explored. If so, we had the current node to the explored node list and its successors to the front of frontier. Otherwise, we move to the next node in the frontier. We push them to the front to encourage consecutive execution. We do so until we reach the end of the frontier. If we reached the end and there are still unexplored nodes, we change the subgraph type to the other backend. Otherwise, we are done.

We explore the graph this way, packing as many nodes together and swapping backends when necessary. Once no more nodes can be added to a given group, we create the corresponding subgraph and move on to the next. Now that the graph is divided into subgraphs, we will see how they are created.

4.3 Object Oriented Approach for Common Subgraph Interface

In order to offer a common interface for subgraphs of different types, we take an object oriented approach. We first define an abstract base class that defines a set of function that need to be overwritten by its subclasses (simply `create()` and `run()`). The base class also takes care of some common initial processing such as determining the input and output nodes from the original graph and given node group to from the subgraph. We then create two specialised subclasses,

one for each backend.

The subclass responsible for ONNX subgraphs is relatively easy. It creates a new ONNX model from the original one that contains only the given group of nodes. We run through the node list, copying the nodes and their inputs and parameters. We also copy the outputs from the original graph if the subgraph contains any. Once the copy of the existing elements is done, we need to add the subgraph's new inputs and outputs. As we saw, ONNX graphs do not contain structures for intermediate activations. For the new inputs, we use the name and dimensions of the outputs of preceding subgraphs. For new outputs, we use the shape inference tool of ONNX to determine the outputs' dimensions. For the execution, we simply run the newly created ONNX model on the native `ONNXRuntime`.

Creating AIX subgraphs is obviously more complicated. In order to map the simple operators of ONNX into complex AIX layers, we pass through an intermediate graph. A single AIX layer can perform a convolution, normalisation, activation and pooling. Nodes of the intermediate have a field for each of these categories plus a misc field to store more particular operation such as residual layers that cannot be merged. We run through the list of ONNX operators that are to form the subgraph and try to add them to a buffer intermediate node. For each operator, we determine to which category they belong and check if they can be merged with the buffer node. If so, we add it to the appropriate field of the buffer node and update its list of predecessors and successors. If the operator cannot be merged (start of a new layer, non consecutive operator, etc.), we add the buffer node to the intermediate graph and create a new node to fill. Due to the way we explored the graph in the earlier stages, operators that can be merged will already be consecutive in the node list so we can simply move through the list in order. While adding the operators to the intermedi-

ate nodes, we keep track of the predecessors and successors with their ONNX indexes. Once we added all the operators to the intermediate graph, we run through the nodes to set the predecessor and successor lists to AIX indexes. At this stage, we have an intermediate graph formed of AIX-like nodes ready.

We can now convert each intermediate node into a AIX layer. We run through the intermediate graph one more time. For each node, we extract the parameters from the ONNX operators they are formed from. We then use the functions defined in our custom module to create AIX layers and define their behaviour by transferring the needed parameters. In some cases, parameters need to be adjusted to match AIX's requirements (prioritized dimensions for padding for example). For each layer, we compute the output dimensions from the input knowing the behaviour of different operations. We also define the subgraph input and output layers and fix addresses in memory where they should read from and write to. AIX quantizes weights and activations to further speedup execution. Float32 values are quantized to int8. Obviously, this reduces the precision of the computation as it limits the number of values we can represent for a given range. To reduce the precision loss, we calibrate the AIX graph. We create so called calibration tables by running a sample of input images and saving the min and max values of each activations. With this information know in advance, we need to represent a smaller range and can thus decrease the gap between quanti. If we can represent 255 values, it is easy to understand that we can represent values between 0 and 5 more precisely than between 0 and 100 for example. The AIX graph is now ready to be compiled into a command graph for the hardware.

4.4 Handling the Data Flow

All subgraphs are now created and ready to be executed independently. We use Python's dictionary structure to properly forward the data between subgraphs. Each subgraph keeps a list of its input names and output names. So if we use these names as keys to the items of the dictionary, we can easily keep track of what is saved where. We store the data as NumPy multidimensional arrays. We give each subgraph a dictionary containing its needed inputs and the subgraph returns its outputs in the same way. The way we explored the graph initially and created the graphs make the inputs will always be ready in time if we execute the subgraphs in order.

4.5 Working with a Custom Tool under Development

AIX is still under development. It is an in house product designed by SKT. Documentation and clear explanations were therefore very difficult to come by. To understand how to create AIX graph, I first went through the translation from darknet to AIX. Since these two graph structures were very similar, it only gave me partial guidance on how to proceed in the case of ONNX. As stated before, AIX is fairly restrictive. However, these requirements are not properly defined. Creating this translator required a lot of trial and error. The only way to find out I was not meeting a condition was at runtime, when the compilation of the AIX graph to the command graph aborted due to an error. I then had to look into the simulator's source code to find out what the requirement was. In addition to this lack of transparency, the API changed drastically a few times during the project requiring to basically start all over. It is safe to say that the development of this ONNX translator was done in less than optimal conditions.

Chapter 5

Test Setup

5.1 CIFAR10

CIFAR-10 [11] is a relatively simple classification dataset. It is composed of 60000 32×32 images. The images are labelled with one of the 10 exclusive classes. The classes and a image sample can be seen in figure 5.1.

We chose this dataset for its simplicity and the ability to quickly design and train a network that solves it fairly well. The prediction part is necessary to evaluate our work as we simply try to reproduce the same results as an ONNX run. But while comparing the geometric distance between the output tensors might be a more accurate criteria, it does not really speak to us. So we decided to show the comparison of the top-1 predictions of the same network run on AIX and ONNX over a sample.

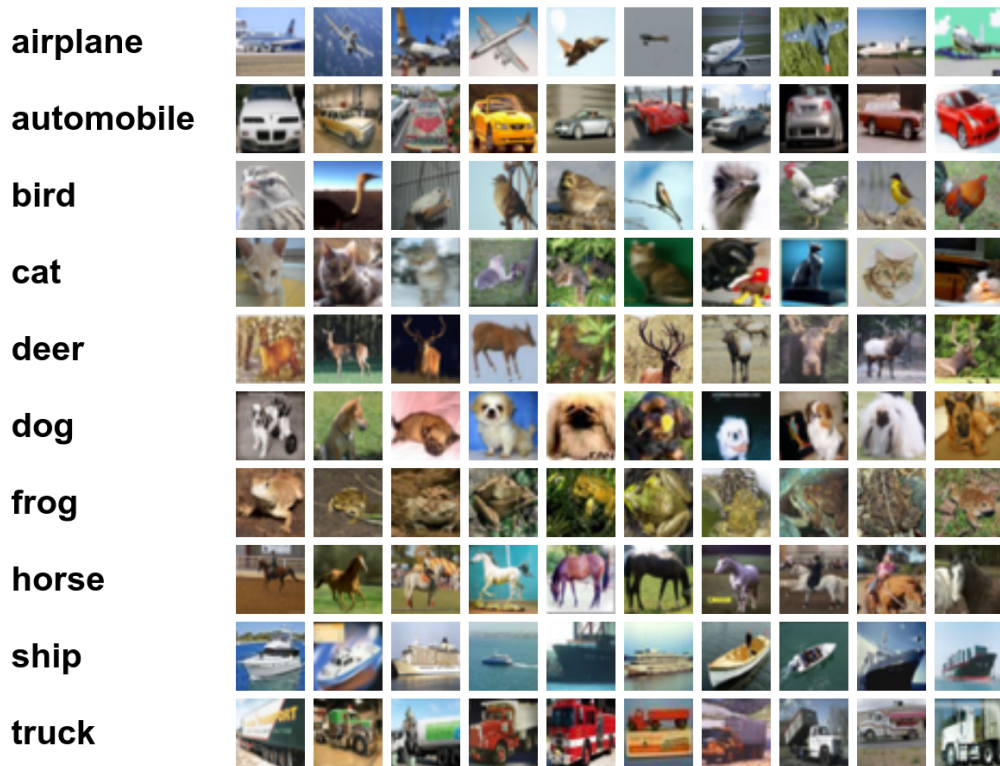


Figure 5.1: CIFAR-10 sample.

5.2 Prototyping and Converting Test Networks

As stated, ONNX is not a prototyping framework. It focuses on conversion and execution. We used Pytorch to design and create our networks. Once the network ready, the conversion to ONNX is easy since there are built-in Pytorch functions for that. We designed a set of test networks of increasing complexity but for the reason stated above, we will here talk about the one we designed for CIFAR-10.

While prototyping this network, we made the structure so that we could make an equivalent network in Darknet for debugging purposes. The network is composed of 4 standard stages of convolution, normalisation, ReLU and max-

pooling. The last standalone convolution replaces a fully-combined layer when combined with the average pooling operator. The activation is reshape into a 2D array $[B, 10]$ where B is the number of batches. We know have a value for each class. The softmax function normalizes the activation so that the sum is one. The output results can then be interpreted as the predicted likelihood how the picture to belong to a certain class. After training, we reach a 70% top-1 accuracy. Eventhough the ONNX prediction does not need to be right for us to evaluate the AIX output, it does make things more interesting.

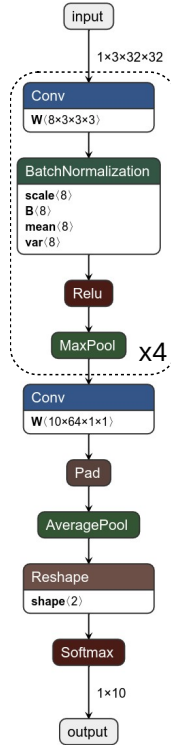


Figure 5.2: Test network.

Chapter 6

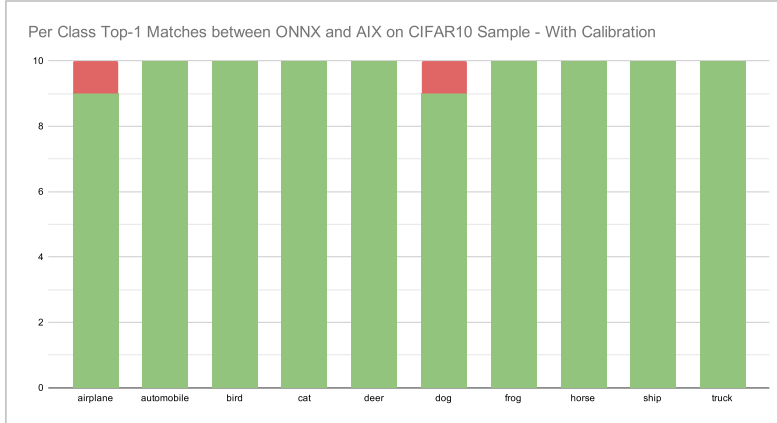
Results

The results of the execution of the CIFAR-10 sample on AIX and ONNX are shown in figure 6.2. The presented sample contains 100 images, 10 for each class. The class of the images is indicated above each list of indices ranging from 1 to 10. We then have the prediction of the network run on AIX and ONNX. We show the top-1 predicted class and the confidence of the prediction.

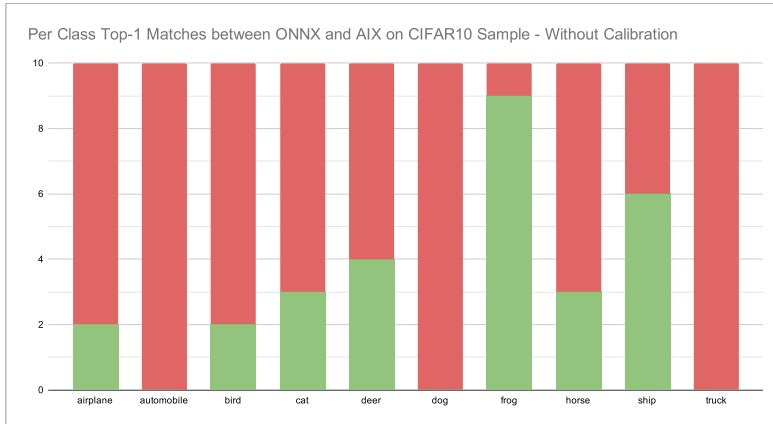
A green colored index cell means that the ONNX and AIX execution predicted the image to be in the same class. Red indicates a mismatch. Blue cells indicate that the top-1 prediction is correct. Yellow means an incorrect prediction. These two latter criteria are not prevalent however. Our prototyped network does not aim to be very accurate. The goal of the project is simply to reproduce ONNX results as best as possible when run on AIX. In that regard, we can see that we perform well. 98% of the images are classified identically. In addition to that, the confidence rates are also very close.

You might ask why results are different in the first place. It is due to the fact that AIX quantizes the networks parameters and the activations. As we men-

tioned earlier, the error due to quantization can be greatly mitigated through calibration. Indeed, without calibration, we only reach a 29% match rate between ONNX and AIX executions. The match rate comparison with and without calibration can be seen in figure 6.1



(a) ONNX/AIX match rate on sample with calibration.



(b) ONNX/AIX match rate on sample without calibration.

Figure 6.1: Effects of calibration on the translation’s match rate. Green means AIX and ONNX classified an image identically. Red indicates a mismatch.

	Top-1 class	Confidence	Top-1 class	Confidence		Top-1 class	Confidence	Top-1 class	Confidence
airplane	AIX		ONNX		automobile	AIX		ONNX	
1	airplane	100	airplane	100	1	automobile	99.93	automobile	99.93
2	airplane	99.96	airplane	99.94	2	automobile	98.91	automobile	98.68
3	airplane	99.99	airplane	99.99	3	horse	47.61	horse	43.3
4	airplane	100	airplane	100	4	automobile	94.67	automobile	91.34
5	airplane	99.95	airplane	99.92	5	automobile	100	automobile	99.99
6	airplane	97	airplane	97.38	6	truck	98.33	truck	98.97
7	airplane	99.98	airplane	99.98	7	automobile	99.94	automobile	99.95
8	horse	61.16	horse	55.32	8	automobile	99.89	automobile	99.86
9	airplane	57.09	ship	69.42	9	automobile	84	automobile	80.81
10	airplane	100	airplane	100	10	automobile	99.97	automobile	99.97
bird	AIX		ONNX		cat	ONNX		ONNX	
1	bird	72.44	bird	80.45	1	deer	99.43	deer	99.46
2	airplane	64.14	airplane	68.1	2	dog	65.45	dog	67.23
3	frog	64.74	frog	57.51	3	deer	74.31	deer	68.34
4	bird	99.99	bird	99.99	4	ship	73.98	ship	67.56
5	frog	77.13	frog	76.71	5	deer	99.06	deer	99.13
6	bird	100	bird	100	6	deer	99.44	deer	99.35
7	bird	97.98	bird	98.44	7	frog	66.1	frog	69.76
8	horse	66.06	horse	61.43	8	bird	78.53	bird	73.85
9	bird	100	bird	100	9	frog	100	frog	100
10	bird	99.98	bird	99.99	10	bird	48.6	bird	50.94
deer	AIX		ONNX		dog	AIX		ONNX	
1	deer	98.33	deer	99.02	1	dog	100	dog	100
2	deer	99.99	deer	99.99	2	bird	99.22	bird	99.22
3	deer	100	deer	100	3	dog	70.56	dog	79.85
4	horse	75.89	horse	75.19	4	dog	100	dog	100
5	deer	99.02	deer	98.95	5	dog	99.72	dog	99.79
6	deer	99.98	deer	99.98	6	frog	50	dog	55.93
7	horse	87.47	horse	78.92	7	dog	99.53	dog	99.61
8	deer	99.92	deer	99.93	8	dog	100	dog	100
9	deer	100	deer	100	9	dog	99.64	dog	99.77
10	frog	90.95	frog	85.54	10	horse	99.97	horse	99.95
frog	AIX		ONNX		horse	AIX		ONNX	
1	frog	99.28	frog	98.49	1	horse	99.99	horse	99.98
2	frog	99.92	frog	99.83	2	horse	99.99	horse	99.98
3	frog	99.97	frog	99.99	3	horse	100	horse	100
4	frog	100	frog	100	4	horse	98.79	horse	98.78
5	frog	99.99	frog	99.98	5	horse	99.65	horse	99.51
6	frog	100	frog	100	6	horse	100	horse	100
7	frog	100	frog	100	7	horse	96.96	horse	95.37
8	frog	99.98	frog	99.98	8	horse	82.09	horse	74.65
9	frog	99.92	frog	99.87	9	frog	46.6	frog	58.38
10	frog	99.16	frog	98.71	10	deer	74.15	deer	74.69
ship	AIX		ONNX		truck	AIX		ONNX	
1	ship	100	ship	100	1	truck	91.27	truck	94.88
2	ship	89.02	ship	93.59	2	truck	99.88	truck	99.82
3	ship	99.98	ship	99.98	3	truck	94.29	truck	91.98
4	ship	99.08	ship	99.03	4	truck	99.79	truck	99.69
5	ship	59.91	ship	60.88	5	truck	68.57	truck	72.95
6	ship	96.14	ship	98.26	6	ship	99.99	ship	99.98
7	ship	100	ship	100	7	truck	94.47	truck	89.46
8	airplane	99.28	airplane	99	8	truck	99.82	truck	99.7
9	ship	91.17	ship	95.23	9	truck	99.03	truck	98.93
10	ship	100	ship	100	10	truck	99.52	truck	99.72

Figure 6.2: Detailed results

Chapter 7

Conclusion

In the presented work, we explained how we proceeded to extend AIX’s support to ONNX networks. Since the accelerator was initially designed for a very different network structure, transforming ONNX graphs into AIX graphs was not straightforward. We saw how we merged simple ONNX operators into complex AIX-like layers. From there, we interacted with AIX’s API to actually create the appropriate objects, adapting some parameters to meet the accelerators requirements if necessary.

In order to be able to accelerate parts of networks that contain unsupported operators, we adopted a flexible approach. Unsupported operators are executed in the native ONNXRuntime framework on the CPU or GPU why computationally heavy operations can still be accelerated on ONNX. To achieve this, we divide the graph into backend specific subgraphs sharing a common interface for smooth interaction.

We designed a test network trained to solve CIFAR-10’s classification problem. We showed that after calibrating the network to mitigate AIX’s quan-

tization precision loss, we achieve very comparable results when running the network on AIX and ONNXRuntime. Indeed, we reach a 98% top-1 prediction match rate on a random sample.

Bibliography

- [1] M. Ahn, S. J. Hwang, W. Kim, S. Jung, Y. Lee, M. Chung, W. Lim, and Y. Kim. Aix: A high performance and energy efficient inference accelerator on fpga for a dnn-based commercial speech recognition. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1495–1500. IEEE, 2019.
- [2] J. Bai, F. Lu, K. Zhang, et al. Onnx: Open neural network exchange. <https://github.com/onnx/onnx>. GitHub repository.
- [3] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.
- [4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [5] Google. Protocol buffers. <https://github.com/protocolbuffers/protobuf>. GitHub repository.
- [6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

- [7] D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.
- [8] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.
- [9] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.
- [10] W. Jakob, J. Rhineland, and D. Moldovan. pybind11 – seamless operability between c++11 and python, 2017. <https://github.com/pybind/pybind11>.
- [11] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.
- [12] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.
- [13] Y. LeCun, K. Kavukcuoglu, and C. Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE international symposium on circuits and systems*, pages 253–256. IEEE, 2010.
- [14] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al. Mlperf inference benchmark. *arXiv preprint arXiv:1911.02549*, 2019.
- [15] J. Redmon. Darknet: Open source neural networks in c. <http://pjreddie.com/darknet/>, 2013–2016.

- [16] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.
- [17] L. Roeder. Netron. <https://github.com/lutzroeder/netron>. GitHub repository.

요약

컴퓨터 비전 문제는 합성곱 신경망을 통해 가장 잘 해결된다. 이러한 리소스 집약적 알고리즘은 특히 애플리케이션이 수행 시간과 전력 효율을 중요시할 때, 특수 가속기를 사용하여 수행되어야 한다. 이와 같은 가속기들은 성능 향상을 위해 일반성을 저하시켜야 하는 한계를 가진다. 이 논문에서 우리의 대상 하드웨어는 다크넷 합성곱 신경망의 실행을 위해 설계된 SKT의 AIX 가속기이다. 이 논문을 통해 제안하는 방법을 통해 AIX에서 ONNX 네트워크를 유연하게 실행할 수 있으며, 이는 가속기의 지원을 다양한 프레임워크로 확장한다. 우리는 신경망의 그래프 구조를 가속기가 가지는 다른 형태의 구조로 매핑하기 위해 취해야 할 단계와 지원되지 않는 작업을 포함하는 신경망의 부분 가속을 어떻게 달성하는지 살펴볼 것이다. 본 프로젝트에서 제안하는 방법을 통해 AIX에서 ONNX 네트워크를 실행하면 ONNXRuntime에서 수행되는 기본 실행과 매우 가까운 결과를 얻을 수 있다. 실제로, 우리의 결과는 CIFAR10 샘플에서 98%의 top-1 일치율을 보인다.

주요어: 컴퓨터 비전, 심층 신경망 가속, 심층 신경망 번역

학번: 2018-25262