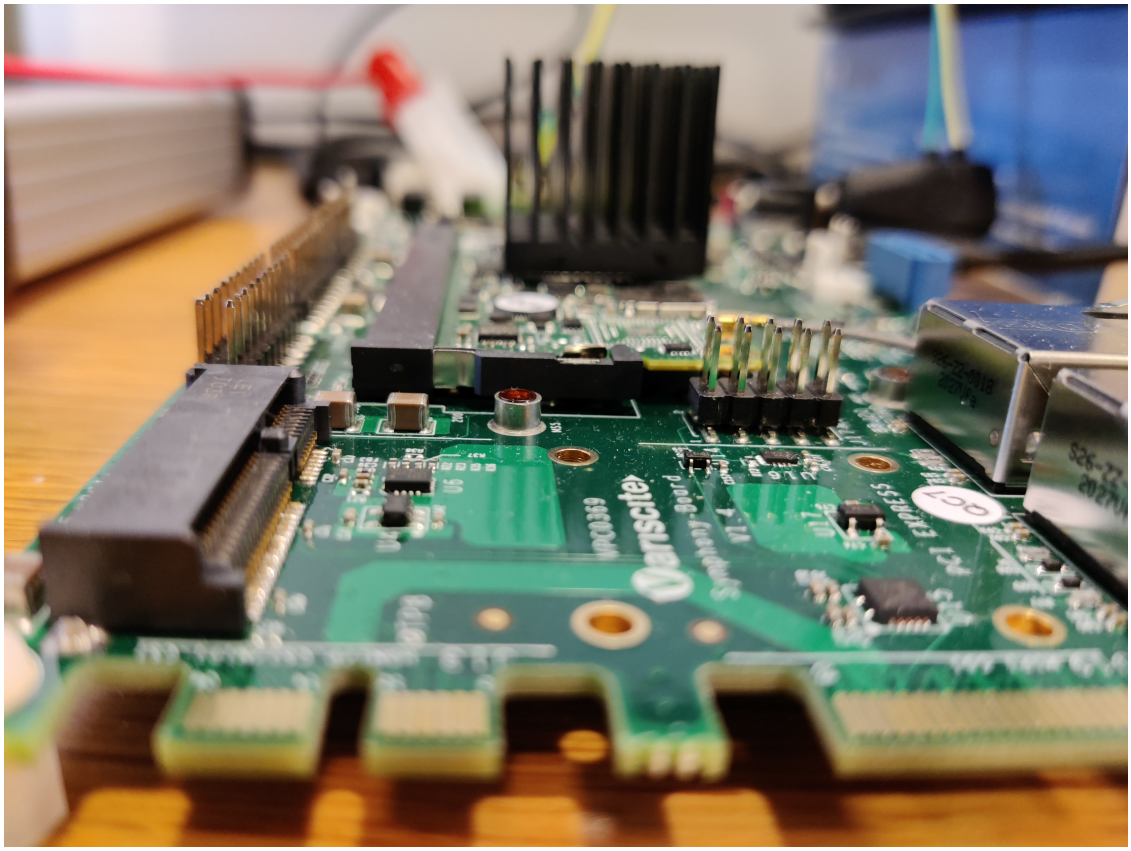
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Neural Networks, Edge Computing or Offloading

A study about how offloading neural network calculations stands in contrast to edge computing for embedded hardware

Master's thesis in High-performance computer systems

ERIK FRENNBORN

ADAM OLIV

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Neural Networks, Edge Computing or Offloading

A study about how offloading neural network calculations stands in contrast to edge computing for embedded hardware

ERIK FRENNBORN
ADAM OLIV

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Neural Networks, Edge Computing or Offloading
A study about how offloading neural network calculations stands in contrast to edge computing for embedded hardware
ERIK FRENNBORN, ADAM OLIV

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Magnus Agren, Prevas AB
Examiner: Risat Mahmud Pathan, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Image of i.MX 8M Plus on the Symphony carrier board that was used in the thesis work.

Typeset in LaTeX
Gothenburg, Sweden 2022

Neural Networks, Edge Computing or Offloading
A study about how offloading neural network calculations stands in contrast to edge computing for embedded hardware
ERIK FRENNBORN
ADAM OLIV
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The use of machine learning and neural networks shows no signs of slowing down in the embedded sector, but since embedded hardware often face performance or energy constraints that heavily limit the computational capacity of applications, it might not always be optimal to perform computations locally on the device. This study compares on-chip computing of neural networks on the embedded hardware *i.MX 8M Plus* with offloading the computation to a remote server supplied with *Intel i9-10850K* and a *GeForce RTX 3070*. This study also investigates to what extent the common compression techniques quantization, pruning and weight clustering affect performance metrics such as latency and energy consumption for embedded hardware. After measuring latency and energy consumption for non-pipelined inferences for 33 network variations we have discovered both a latency and energy consumption threshold where it is more effective to offload rather then computing on the edge device. These thresholds exist since latency for the offloading scenario remains almost constant. The latency threshold of 0,031 s is obtained from offloading via Ethernet to remote GPU, and is strongly limited by the network latency. This inference time could be used as a guideline for development meaning that if the inference time on embedded device exceeds set time, it is probably more efficient to offload calculations to remote server. Other discoveries points towards the conclusion that the NPU of the *i.MX 8M Plus* heavily favors compressed models, showing an average speedup of 188x on the NPU when models are compressed using quantization and pruning.

Keywords: Offloading, Convolutional neural networks (CNNs), Deep neural networks (DNNs), Embedded systems, Edge computing, Energy reduction, Optimization, Compression.

Acknowledgements

We would like to thank our supervisor Pedro Petersen Moura Trancoso for his help and insights around the topic of neural network compression as well as overall guidance during this period. We also would like to thank the people at Prevas especially our advisor Magnus Agren for his mental and technical support along the work.

Erik Frennborn and Adam Oliv, Gothenburg, June 2022

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
AI	Artificial Intelligence
NPU	Neural Processing Unit
DCNN	Deep convolutional neural networks
FPGA	Field-programmable gate array
ASIC	Application-specific integrated circuit
tflite	Tensorflow-lite
RTT	Round Trip Time
EDP	Energy delay product

1
Introduction

The last couple of years machine learning have been a growing topic among researchers and in the industry opening up doors to many new technical solutions in the facial recognition [1] and autonomous driving [2] domains among others. Traditionally many papers on the topic of machine learning has fell into one of two categories, either pure algorithmic where hardware is not a concern, or papers focused on developing hardware to better suit common tasks found in the machine learning topic. Lately a third category called co-design [3] has been growing in interest, which is when the hardware and software are designed together. While hardware is being optimized for machine learning, the neural networks are also adapted to the desired hardware memory, energy and time constraints set on the device. This is a growing subject because it is one of the key elements in order to introduce machine learning to embedded computing [4].

Currently it is common for embedded applications to offload computer heavy tasks to cloud servers where the Artificial Intelligence (AI) models are executed. When only considering the performance metrics that comes with offloading computational heavy tasks to cloud or remote server, there are many arguments for and against offloading computation. In order to perform calculations on the edge device the device needs to be powerful enough to handle the heavy computation in a timely fashion. There are multiple factors that contribute to the decision of whether or not some calculations should be calculated on the device or offloaded. Performance critical tasks as for example real-time person detection in autonomous vehicles should probably not be offloaded since performance and latency could be safety critical in some situations. There are also other factors that can exclude the offloading alternative, some devices do not have reliable internet connection as for example embedded devices in cars or other mobile devices. These devices that are mobile can not rely on internet connection and must work independently and thus requiring on-chip calculation or local offloading to other connected devices. On the other hand some tasks are not heavily performance or safety critical in the same sense and can afford the extra added communication latency.

Offloading limits the application space since the offloading requires internet connectivity, thus introducing latency and privacy concerns. There exists different ways to improve security when offloading, studies like *Privacy Aware Offloading of Deep Neural Networks* [5] have developed methods for improving security for offloading neural networks. If instead the model would be executed on the embedded device these limitations would be resolved and open up new possibilities for embedded devices.

This work have evaluated inference performed on different compute platforms such as on embedded CPU, embedded Neural Processing Unit (NPU) and offloaded CPU and GPU. These different compute platforms will be evaluated on problems in the object classification domain. Specifically deep convolutional neural networks (DCNN), with the expectations that some of the results will be applicable in a larger part of neural network domain. DCNNs have been predominant in the task of object detection and classification of images during the recent years in the aspect of pure classification accuracy. DCNNs has therefore been picked as the result would be relevant to a larger number of applications as for example edge computing. DCNNs also introduce memory constraints since they normally are memory intensive, which is a common problem when integrating large neural networks on embedded devices.

1.1 Background

Embedded devices have a number of limitations that general computing systems lack. Mainly the limited resources, primarily memory size, CPU and GPU performance as well as sometimes limited battery. These limitations means that the commonly used machine learning techniques are infeasible or impractical on embedded devices as they would not be suitable for the device. This is because performing heavy calculations on the embedded device may either too slow to be useful or fall short on other requirements. Additionally as many embedded devices are battery powered, thus making energy consumption of the device of high importance as it would otherwise run out of battery too quickly. These issues gets considerably worse when taking into account that embedded systems often are sensors and often consist of multiple devices deployed over a wide area, because maintenance of such number of devices is costly and tedious, resulting in less application of this kind of technology.

1.1.1 Offloading

Research is also advancing an other related topic which is the topic of offloading. Recent studies [5, 6] on the topic have made major strides in improving the latency and security of offloading between edge devices and cloud servers, with little or none accuracy loss. Even if offloading introduces communication latency sometimes it is still the best option when considering implementation cost as opposed to performance. Edge devices capable of performing on-chip heavy calculations are more expensive thus making it less cost effective to implement edge computing in those scenarios.

1.2 Aim

This work explores existing trade-offs of offloading the execution of neural network models from edge device to cloud computer, more specifically comparing performance metrics such as latency and energy consumption when calculations are performed on embedded CPU, NPU, offloaded CPU and offloaded GPU. The comparison is evaluated on different DCNNs differentiating them primarily by network size and input size in order to gain a broad overview of how differently scaled networks affect the selected performance metrics.

1.3 Scope

This work explores how the common compression techniques quantization, pruning and weight clustering affect performance metrics such as inference latency and energy consumption. Other compression techniques are not evaluated in this work since Tensorflow only support these named techniques to this day. Since the NPU is support the *Tensorflow lite* tooling this decision was made in order to gain a smoother integration between models and hardware.

The energy measuring is only considering the energy consumption of the edge device, meaning that energy from the network infrastructure is not considered. This would not alter the results since only the edge device has potential energy consumption constraints. When deploying an embedded device it is important to estimate the devices' energy consumption since it may run on battery while the networks energy consumption is not free it does not contribute to the edge device performance metrics and may thus be excluded from this work.

The selected hardware is limited to the *i.MX 8M Plus* as the edge device, while the offloading host computer is limited to *Intel i9-10850K* CPU and a *GeForce RTX 3070* GPU with the main reason behind this scope being resources availability.

2
Theory

2.1 Neural Networks

Neural networks [7] can be described as a series of algorithms that aims to imitate a more complex function. Neural networks are made up by a group of neurons, were each neuron applies an input specific weight to a number of inputs and outputs the sum of the weighed outputs. The neurons are usually grouped into three different layers being input, hidden and output layers, where each of the layers has its own contribution to the neural network. A common use case for neural networks is image recognition and classification, where in the case of image classification the input layer is traditionally represented by one neuron per pixel that holds the grey scale or RBG values of each pixel of an image. The hidden layers are not mandatory but exist in order to gain a better classification accuracy. This is because splitting up the neurons into layers adds higher levels of abstraction to the neurons, since the input of the second layer neurons is the first layers outputs. An example of this is where the neurons of the first layer represents the color value of each pixel, the second layer may weigh the inputs to represents edges or sections of the image. The output layer is traditionally used as classification label where the number of neurons often match the number of classifiable classes, with the exception of binary classifiers where a single output neuron could do the same work outputting either one or zero for the different labels. For example a network meant to classify handwritten numbers should have ten output neurons where each symbolises the probability of that class for the image.

2.1.1 Convolutional Neural Networks

A convolutional neural network[8] is a neural network where some hidden layers are constructed with whats called convolutional layers. A convolutional layer is made up out kernels also known as filters. The kernel is usually a smaller part of the image often consisting out of a few pixels each. The product of the kernel and respective image pixels are connected instead of having all neurons of the previous layer connect to each neuron in the layer, this enables neural networks both to learn that neighboring pixels have larger contributions to other nearby pixels as well as reduce the parameter count of the network.

This work will evaluate the convolutional neural networks listed in Table 2.1. These neural networks are sometimes called deep convolutional neural networks, this is because they contain multiple of hidden layers.

Table 2.1: Selected neural networks with their network depth and parameter count.

Model	Input size	Network depth	Network size (#parameters)
Inceptionv3	75x75	189	22353028
Inceptionv3	128x128	189	23925892
Resnet 50	32x32	107	24137956
Resnet 50	75x75	107	28332260
Resnet 50	128x128	107	32002276
MobileNet	32x32	105	3516964
MobileNet	75x5	105	4303396
MobileNet	128x128	105	7449124
VGG16	32x32	16	14871716
VGG16	75x75	16	15264932
VGG16	128x128	16	16837796

2.1.2 Neural Network Compression

In order to improve the performance of AI on embedded devices studies have been made on neural network compression, these papers state and evaluate techniques that manipulate neural networks to better fit to embedded devices [9, 10]. Compression of neural network has proven to reduce the size of network models while keeping or with minor losses in accuracy. Other papers also compare different models of different sizes against each other [11] proving that a comparative study is valuable for both future research and providing support for real application scenarios. The most frequently used compression techniques are quantization, weight clustering (also called weight sharing) and pruning [12]. These compression techniques are independent but can be utilized in combination for greater compression.

Quantization

Quantization [13] is a technique that reduces the number of bits used to represent each weight in the neural network. A common example is reducing the bit weight size from 32 bit floating point values to 8 bit integers. This conversion is one of the most commonly used compression techniques for neural networks and reduces the size linearly to the weight bit size. This example conversion from 32 bit to 8 bit achieves a compression rate of 4x. In addition it is common to covert the standard float weights into the integer datatype, since integers is a much more efficient and less complex datatype than floats. As shown in "Loss-aware weight quantization of deep networks" [14], quantization as low as 3-bit weights results in as and even more accuracy compared to full-precision network.

Pruning

Pruning [15, 16, 17, 18] is a compression and regularization technique, apart from quantization and weight clustering, pruning actively changes the network behavior. There are many ways to decide which weights to keep and which to prune, one of the most common pruning strategies is to prune the weights with the smallest values. Intuitively this makes sense since removing the weights that contribute the smallest amount will result in the minimum network behavior variance. Since pruning also is a regularization technique it can sometimes even improve the accuracy because it helps to reduce overfitting. Overfitting can occur for a number of reasons, either when a model has been trained on a non general dataset or simply memorized the training dataset. The problem that overfitting introduces is that the model performs poorly on new previously unseen data making the model perform worse when deployed. There are multiple ways to combat overfitting, model pruning being one of the more prominent methods [19].

Weight Clustering

Weight clustering [12] recognises weights in the same layer that are close enough to each other in value and normalizes and reuses the centroid values for all weights in the cluster, this is visualised in Figure 2.1. The centroid values are commonly calculated with the average weight values within a interval of the maximum and minimum weights inside a layer. The weights are replaced by cluster index values usually consisting out of one to 4 bits depending on how many bits needed to describe the number of clusters. The centroid values are then possibly fine-tuned on the training dataset in order to minimize the potential performance loss obtained from reducing the weight bit size. This technique is powerful in combination with quantization since reducing the bit weight size reduces the amount of potential different weight values resulting larger utilization of the clustering technique [12]. This technique does not provide linear compression rate and works better the larger networks is, since there are more weights that can be clustered together. The clustering technique introduces some extra allocated memory needed to store the centroid values, but together with specialized run-time or compiler software and dedicated machine learning hardware further compression can be achieved and this extra memory becomes negligible [20, 21].

2.2 Neural processing unit

The execution of neural networks consist of highly parallel operations as such the executuion accelerates well on GPUs. Along with the advances in the different fields of embedded neural networks such as compression and offloading, newer hardware architectures are also emerging. Different architectures [22, 23] propose different approaches to the problem of deploying neural networks efficiently on embedded devices. All implementations of Neural Processing Units (NPUs) are meant to optimize the instructions and data flow common found in neural networks, such as multiply add instructions, in order to speed up the inference time and lower energy

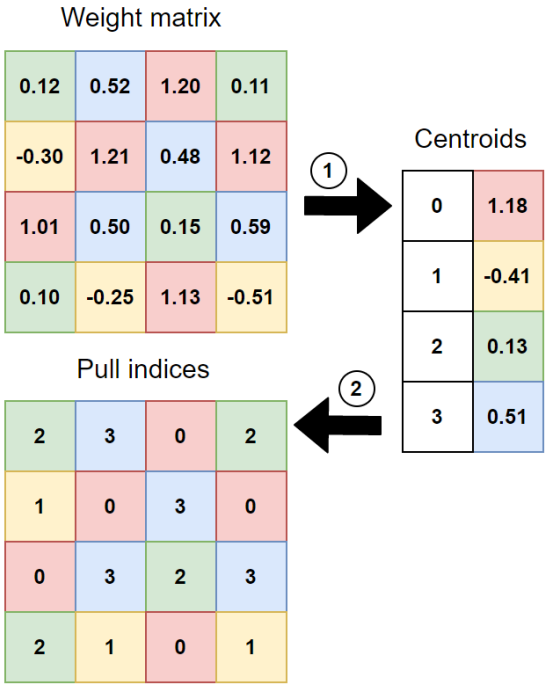


Figure 2.1: Visualization of the weight clustering concept.

cost for computation.

2.3 Offloading

Offloading is a concept that builds on separating the data collection from its processing, meaning that parts of the computation are executed on another device than the one that produced the input for the computation. It exists different and more advanced offloading techniques [24], which improves the performance of execution latency and throughput. Offloading allows resource limited devices to use much larger models than otherwise possible by hosting these models on stronger devices, this of course comes with downside of needing to communicate with remote host device for each inference.

2.4 Network model

The latency model for offloading can be split into three different contributing factors being the following.

- The time needed to send input data to the offloading machine, in this case the image.
- The inference latency on the remote server or computer.
- The time to return the result.

Likewise the energy consumption model breaks into similar parts.

- The energy needed to transmit the data,

- The energy used to receive the result
- The energy consumed by the CPU while it is waiting for the result.

Table 2.2: Variable description

s_{input}	Input size (B)
s_{result}	Result size (B)
t_{OINF}	Offloaded inference time (s)
$t_{latency}$	Network lactency (s)
P_{idle}	Embedded CPU idle (w)
$E_{Network}$	Network energy consumption (j/Byte)
B_{Upload}	Network bandwidth upload (Byte/s)
$B_{Download}$	Network bandwidth download (Byte/s)

The total network delay can be calculated using the formulas 2.1 and 2.2

$$t_{RTT} = 2t_{latency} \tag{2.1}$$

$$t_{network\_tot} = t_{RTT} + \frac{s_{input}}{B_{upload}} + \frac{s_{result}}{B_{download}} \tag{2.2}$$

The network latency then consist of the Round Trip Time (RTT) and the time to send and recessive. This is done for both Ethernet and wireless. The network latencies for offloading to CPU or GPU is created using the formulas 2.3.

$$t_{OINF\_tot} = t_{OINF} + t_{network\_tot} \tag{2.3}$$

The energy consumption of the embedded device for the offloading case can be calculated by the sum of the energy used by the network and the energy consumed by the CPU/NPU while waiting for a response. Shown in formula 2.4

$$E_{network\_tot} = \frac{s_{input}}{E_{Upload}} + \frac{s_{result}}{E_{Download}} \tag{2.4}$$

The total energy consumption for offloading to CPU/GPU is calculated with 2.5, starting with energy used for the communication together with the idle waiting time.

$$E_{OINF\_tot} = t_{OINF\_tot} * P_{idle} + E_{network\_tot} \tag{2.5}$$

The corresponding calculations for the non-offloaded case is trivial since the total latency and energy consumption are simply the measured latency and energy consumption for the embedded CPU and NPU respectively.

3
Method

3.1 Hardware

The specific hardware that will be used for this work is an *i.MX 8M Plus* [25] as the edge device. The *i.MX 8M Plus* is an device optimized for machine learning and neural network applications, which is equipped with an *Quad Arm Cortex A53* CPU that includes an NPU that runs at 2.3 Terra operations per second (TOPS), however little other information exist about the design of the NPU. The architecture of the *i.MX 8M Plus* is displayed in Figure 3.1

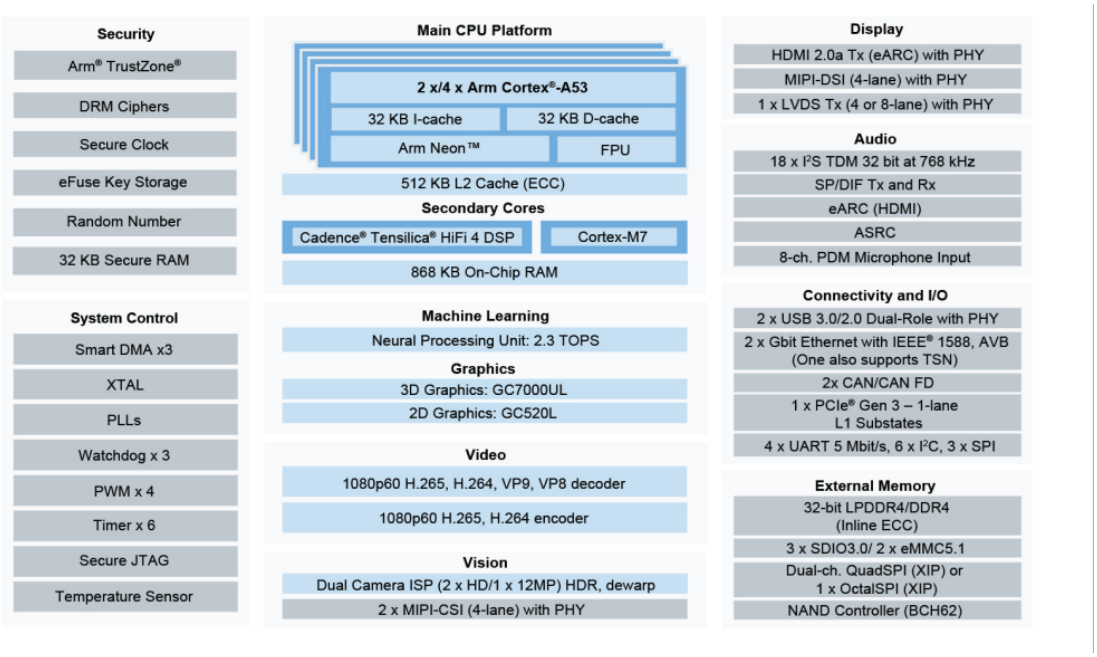


Figure 3.1: Architecture diagram of *i.MX 8M Plus*

As for the offloading hardware a personal computer supplied with an *Intel i9-10850K* CPU and a *GeForce RTX 3070* GPU is used.

3.2 Environment Setup

In order to run the models on the *i.MX 8M Plus* board it was necessary to install an image while the provides support for the NPU. The image was the nxp-demo-experience image for this platform with it a Linux distribution and the required

software needed to evaluate TensorFlow-lite (tflite) models. The image was built using the Yocto build system [26] and the manifest can be found in appendix B. The *Tensorflows benchmarking tool* had to be built from source [27] in order to work as expected on the offloading host computer. When rebuilding the tool additional flags had to be specified in order for the tool to correctly utilize the GPU for computation. The command 3.1 includes the complete command with necessary flags in-order to rebuild the tool.

Listing 3.1: Build command used to build Tensorflows benchmarking tool with GPU support.

```
bazel build --opt=-DCL_DELEGATE_NO_GL
--copt=-std=c++17
--copt=-DCL_TARGET_OPENCL_VERSION=220
--copt=-DMESA_EGL_NO_X11_HEADERS
--copt=-DEGL_NO_X11
tensorflow/lite/tools/benchmark:benchmark_model
```

3.3 Neural Networks

The models used for this experiment are pre-trained models without top classification layers that consist out of normal fully connected layers with an output shape representing the number of classes to classify, downloaded from the Keras API [28]. The motivation behind having pretrained models is to imitate real world scenarios as much as possible, and the fact that transfer learning [29, 30] has a growing trend of application it was appropriate to use pretrained models. The selected models are *VGG16* [31], *ResNet50* [32], *InceptionV3* [33], and *MobileNet* [34]. These networks were selected as they provide a wide spectrum of network architectures, input size, and network sizes. As such they provide a good reference for how the networks parameters effect performance. All models are pretrained on ImageNet [35] and combined with two non trained dense classification layers. The added top classification layers are needed since standard top layers from Keras API only support one input shape per network.

The models uses the *CIFAR100* [36] dataset as a representational dataset, since a sample dataset is needed as the chosen compression techniques require either fine tuning training or a representative dataset. *CIFAR100* is a labeled subset of the 80 million tiny images dataset [37] consisting out of 100 labels with 600 images for each label. These models were then compressed using a combination of compression techniques, them being pruning, quantization and weight sharing in order to get a good compression rate. The pruning and weight sharing required fine tuning that was performed on the original dataset used for initial retraining of the network. In order to perform full eight bit integer quantization a unlabelled representative dataset was needed, for this a test subset from *CIFAR100* was used. Both the compressed and uncompressed versions of each model was evaluated to the extent of hardware support. This was done in order to gain a better understanding of how much the compressed networks differ in execution interference on the different hardware processors.

3.3.1 Hyperparameters

The compression techniques can be specified with hyperparameters in order to tweak the aggressiveness of the compression. The hyperparameters used by the compression techniques in order to generating the models are described below.

The quantization technique used for the compressed networks is full int quantization [38], meaning that the default 32 bit float weights are converted to 8 bit integer values. This is because it first most shrinks the model bit size by 4x and convert weights into a less computational datatype. 8 bit quantization was selected since it is currently the smallest quantization variant supported by *Tensorflow Lite* [39], making this closer related to what a real use case could look like.

The pruning technique uses a pruning schedule [40] that uses constant sparsity of 50%, meaning that half of the weights in each layer are pruned. As such that all pruned networks will be evaluated with the same amount of pruning. Constant pruning might not always be the best technique, as it might prune none redundant weights there by reducing the classification accuracy. Constant pruning entails that all networks are compared against each other fairly and was thus the choice of method for pruning.

For the clustering [21] compression technique the number of clusters per layer was set to 16, meaning how many different lookup values the weights can be represented by. 16 was selected as the number of clusters since it provides a large enough range of cluster to be efficient in a real possible scenario. The way that these cluster values are obtained can be specified by its own hyperparameter *cluster centroids init* where the alternative *LINEAR* is used, meaning that the cluster centroid values are evenly spaced between the maximum and minimum weights of that layer.

3.4 Modelling and Evaluation

In order to simplify the experiments by removing the need to build a framework for offloading. The evaluation was done using a theoretical model of the system rather implementing a system for the offloading. The model used is described in Network model 2.4. The latency and energy consumption per inference was created and measured experimentally for each of the models. While the network speed and latency are based on aggregated data [41] for both wired and wireless connections. These results in latency of 10 ms and 29ms for Ethernet and wireless respectively. The radio gives a download speed of 3.53MB/s and 1 MB/s upload as well as a energy efficiency 110nj/B download and 139 nj/B upload. While the Ethernet give a download speed 7,13 MB/s and 3,04 MB/s upload and a energy efficiency 128nj/B and 57nj/B respect upload and download.

Tensorflows implementation of weight clustering normalizes and clusters the weight values to the centroid values. This means that the size of the network is practically the same as the non clustered model. According to Tensorflow common compression techniques like zip can be used to visualize its effect, Tensorflow also state "To further unlock the improvements in memory usage and speed at inference time associated with clustering, specialized run-time or compiler software and dedicated machine learning hardware is required" [20], with this it became reasonable to test versions without weight clustering active, and investigate if the clustering technique brings any unexpected consequences when applied without specialised hardware.

Measuring the inference latency on the embedded device was conducted using *Tensorflows benchmarkings tool* [42]. The tool measure the distribution of the latency for inferring a random input, where the average inference latency was used. The benchmark allows the user to specify among other things the number of inferences, number of threads to use, and if to use the NPU or GPU.

In order to measure the energy consumption of the embedded device a Joulescope [43] was connected to the power feed port. The Joulescope is a tool that is connected in series with the device under test on both the positive and negative leads. Then the device measures several metrics, among which voltage and amplitude, which then can be integrated of period of time. The test bench used in this work consist of a Velleman LABPS3005DN power supply with an output of 12v and max 1.5A, which is connected via the Joulescope and lastly to the i.MX 8M Plus card, as shown in Figure 3.2.

In order to calculate the energy per inference, the Tensorflows benchmarking tool was executed to for 150 seconds in order to get a large sample and reduce the impact of noise. When the benchmark completes, it provide the number of inferences completed. While the benchmark executes the Joulescope records the power consumption, which the Joulescope later uses to calculate the energy consumption over the inference time. It worth noting that only the time use of inference is measured, meaning that warm up periods are ignored. This choice was made as the NPU is

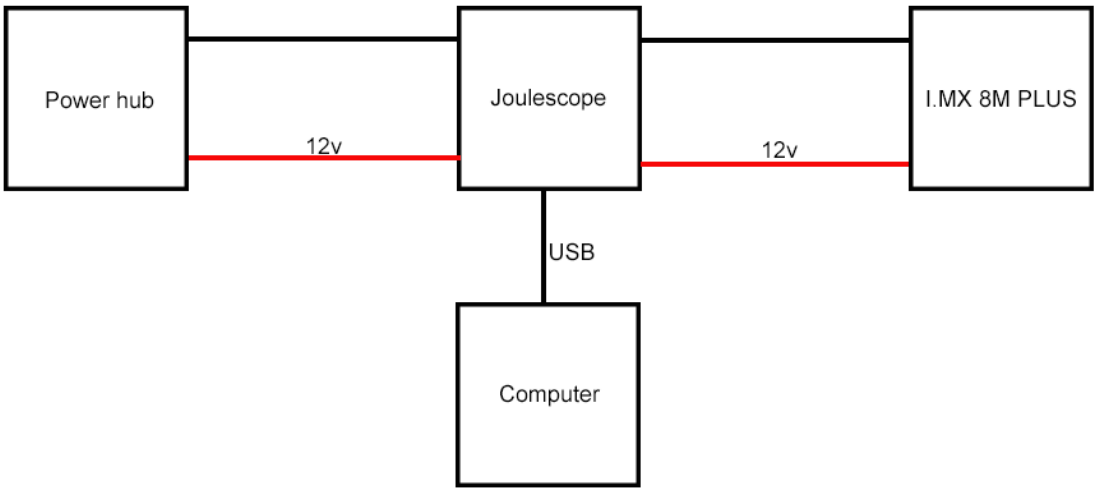


Figure 3.2: Testbench setup

slow to load new weights and will mostly in a real world scenario be loaded once during initiation. Hence including the warm up energy would result in unfairly weighed results against NPU usage. Finally the energy over the inference period is divided by the number of completed inferences, resulting the energy per inference metric.

In order to further profile the AI models, performance counters was used. The Linux tool perf [44] was used to access the counter and allows monitoring of valuable metrics during inference. The metrics that were observed in this study are L1a dcache loads and misses, page-fault, and instruction count. These metrics was selected as they provide an insight into the performance of the memory system and show how many instructions that were executed. The evaluation of performance counters for a model can performed as seen below.

```
perf stat -e {Metrics} ./linux_aarch64_benchmark_model
--graph={model_filename} --num_threads={num_threads}
--use_nnapi={use_nnapi} --warmup_runs=0 --num_rubns=250
```

In this work a python script was used to automate the evaluation, the script can be found in the accompanying files with the name *perf_script.py*

3.5 Known Limitations

A number of limitations have been noticed that may skew the results away from reality. There are two primary such limitations have been identified in this experiment. Due to the energy being measured is the entire power consumption for the device. This is because the platform used to execute the model is a Linux system, so some background tasks had be running and effecting the energy usage. This factor should have limited impact on the relevant comparisons due to the long sample times, as such the energy consumption of background task average out to have lite impact on the final results.

Another potential source of error is that the network parameters are estimated and as such could be erroneous. This is believed to be fair estimations since they are relied on data and other sources, but they are still classified as error sources since they are not real measured values.

4
Results

The results featured in this section visualises inference latency, energy consumption as well as Energy Delay Product (EDP) for the tested models together with analysis of potential causes and contributing factors to the obtained results.

Table 4.1: Explanation of Figure legends. *Model name could be any model name for example InceptionV3

Model	Variant	Input dimension
Model name*	(Q/P/W)	(32/75/128)

In this work the models are denoted with the model name follow by the compression techniques used and lastly the input size. The compression techniques are denoted with the first letter in the name of the compression technique, quantization(Q), pruning(P), and weight clustering(W). Additionally the compression technique letters appear in the order the techniques where applied to the model. Lastly the input dimension states the shape of the input image into the networks meaning that a model that ends with 32 takes a 32x32 image as its input. For instance Mobilenet PQ 32, represents the Mobilenet model with an input shape of 32x32 values, compressed with pruning and quantization.

Following the method described in sections Network model 2.4 and modeling och evaluation 3.4 result was obtained for both cases, edge computing and offloading to host. Both cases feature inference time and device energy consumption when both computing locally on CPU or NPU or offloaded to a CPU or GPU.

4.1 Latency

The obtained inference latency from executing a single inference on each listed unit can be seen in Figure 4.1. The units denoted with "Ethernet" or "Radio" are the offloaded cases where the latency of offloading is estimated with different offloading techniques being Ethernet or over WiFi radio. The offloading technique is followed by the computational units denoted "CPU" and "GPU" which reference to the 10 core Intel i9-10850K offloaded CPU and GeForce RTX 3070 offloaded GPU. While the standalone CPU 1-core and CPU 4-core relates to the edge device CPU *Quad Arm Cortex A53* since no offloading technique is used in this scenario. Similarly the edge device NPU is denoted NPU.

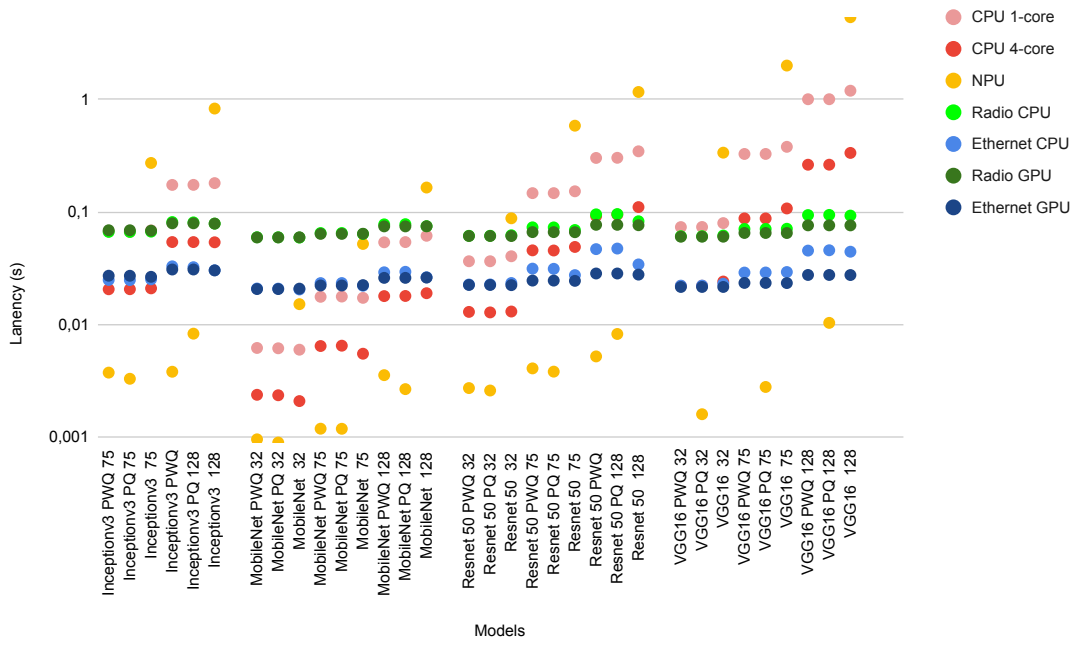


Figure 4.1: Graph of inference latency obtained from executing the inference on different compute platforms. Y-axis is logarithmic.

From analysing Figure 4.1 it becomes clear that the offloading latency remains fairly consistent between models. This is visualised by the green and blue dots in Figure 4.1. These offloading latency's forms two thresholds for radio and Ethernet offloading, each of these thresholds indicate whether or not offloading is the optimal choice for respective offloading technique. The average, maximum, minimum and standard deviation of total latencies are described in Table 4.2 for each offloading scenario. For the Ethernet the threshold in located around 24 ms with a standard deviation of 3 ms. The threshold for radio is located around 68 ms with a standard deviation of 7 ms. The thresholds depends on the models and the input size of the models, however from the results seen in this work a upper bound can be seen. The upper bound was calculated by the maximum latency when offloading to GPU over Ethernet and over radio, visualised with the dark blue and dark green dots in Figure 4.1 respectively. This results in a upper bound of 31 ms for Ethernet and 79 ms for radio, these bounds should give a good estimation for models with similar sized input and design. It is important to note that these threshold are specific to the offloading connection and compute platform used in this work. However the increase in latency is likely to be limited as the offloading latency is dominated by the round trip time as shown in Figure 4.2.

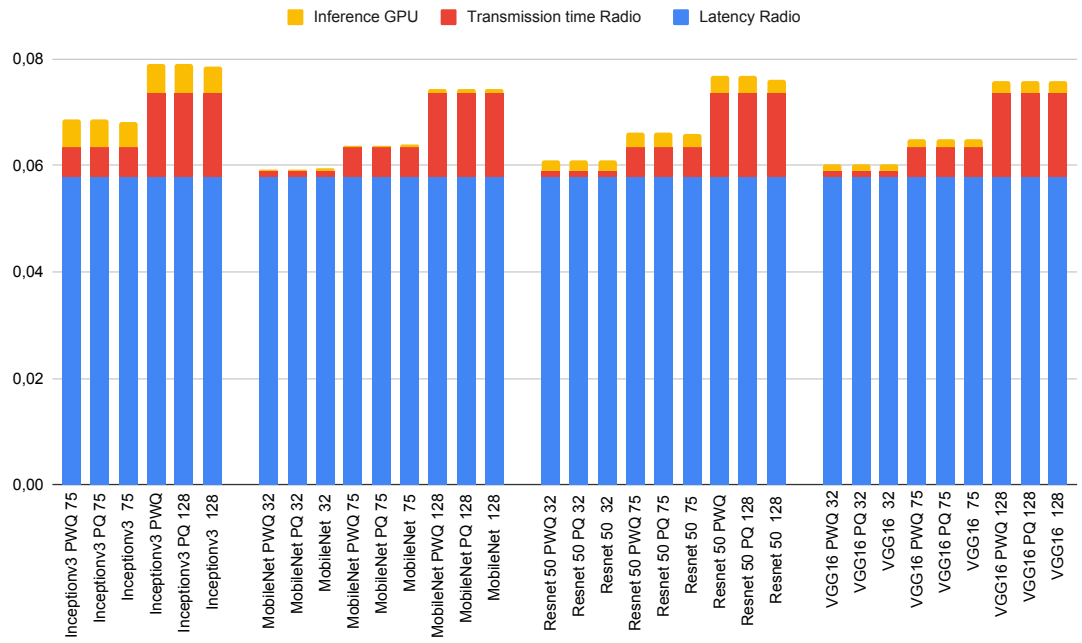


Figure 4.2: Illustration of latency contributions of network communication and execution time of model, using radio to offload to GPU

This pattern exist in most offloading scenarios as can be seen in Figures A.11 and A.12. However some edge cases exist as seen in Figure 4.3, where larger input sizes on the models Resnet 50 and VGG16 are slow on CPUs. Even in these edge cases the latency of the communication remains to be the highest contributing factor of the total inference time indicating that it is a bottleneck even for worst case scenarios.

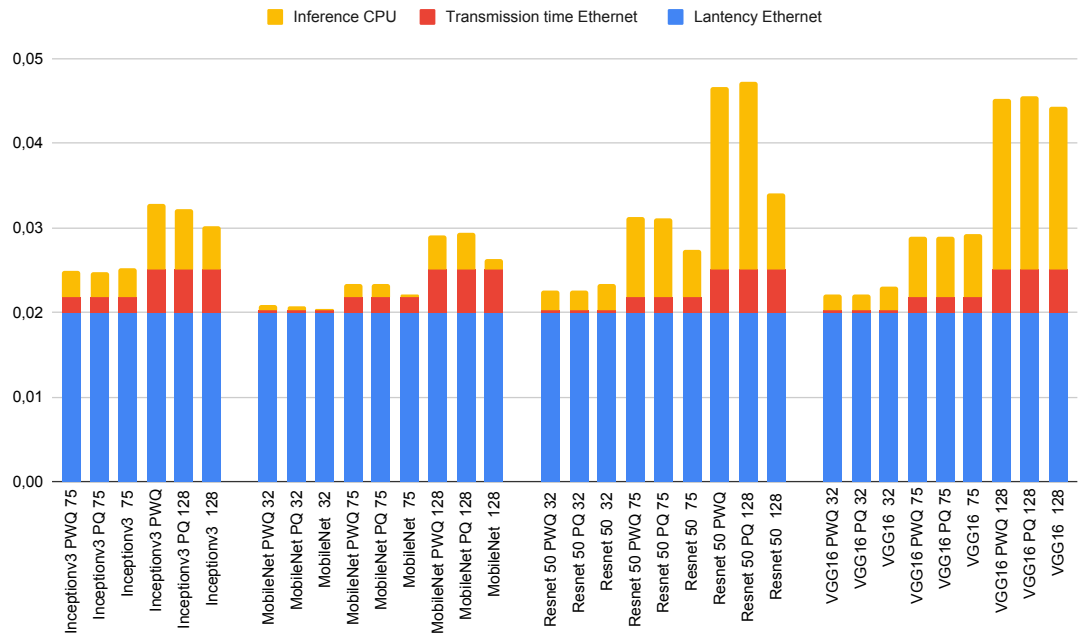


Figure 4.3: Illustration of latency contributions of network communication and execution time of model, using Ethernet to offload to CPU

Further Figures A.14-A.16 visualises the same content found in Figure 4.1 but grouped and isolated by compression techniques. These Figures better visualize the the optimal computation platform for each model variation, since Figure 4.1 may be hard to interpret.

Table 4.2: Threshold latency

	Average (ms)	stdev (ms)	max (ms)	min (ms)
Radio CPU	72	12	96	59
Ethernet CPU	29	8	47	20
Radio GPU	68	7	79	59
Ethernet GPU	25	3	31	21

Noteworthy is that the latency of uncompressed models are penalised on the NPU and suffer from heavy in terms of both latency and energy. The average speedup obtained from compressing the neural network with pruning and quantization is 188x, with a maximum speedup of 716x on the NPU.

4.2 Energy Consumption

Figure 4.4 visualizes the energy consumption of the device for the same cases presented in Figure 4.1. Since it is possible to suspend or enter a lower power mode by suspending the OS when waiting for the response from host in the offloaded cases

the result is supplemented with Figure 4.5 that represents the energy consumption for the same cases with the exception that a new lower idle consumption is used for the offloading cases.

Similar to the latency, two clear thresholds can be seen for both offloading over Ethernet and over radio. For the scenario with higher idle consumption visualised in Figure 4.4 the thresholds for offloading over Ethernet are 75 mJ and over radio are 207 mJ which a 9 mJ and 21 mJ respectively. The energy consumption statistics for the high idle power scenario are collected in table 4.3. While for the scenario with the lower idle consumption visualised in Figure 4.5 the thresholds for Ethernet is around 23 mJ with a standard deviation 3 mJ, and for radio around 32 mJ and a standard deviation of 6 mJ. Then the upper bound for offloading over Ethernet would be 28 mJ and 72 mJ for radio for low idle. The energy consumption statistics for the low idle power scenario are collect in table4.4 While for the higher idle scenario the Ethernet break point 94 mj and 241 mJ for radio.

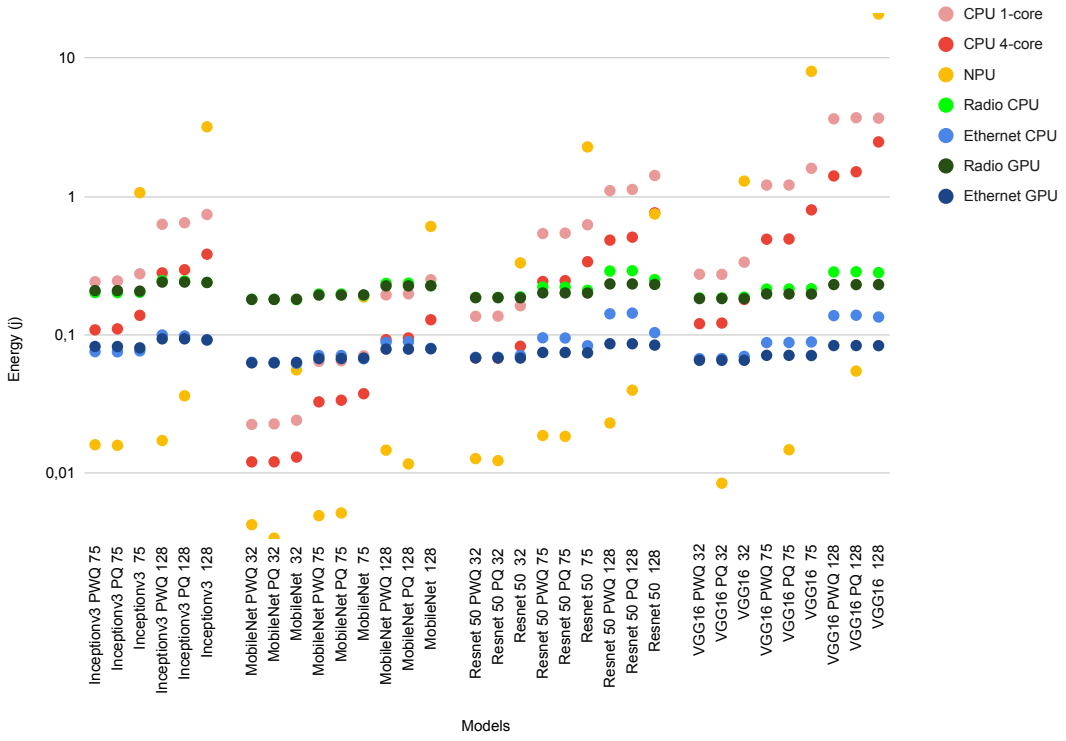


Figure 4.4: Graph of energy consumption obtained from measuring the power over the execution of the inference on different compute platforms. Threshold for radio at 207 mJ and Ethernet at 75 mJ. Y-axis is logarithmic.

These two thresholds are dependent on the idle power of the edge device, in this case lowest idle power observed in this study is about 1 watt.

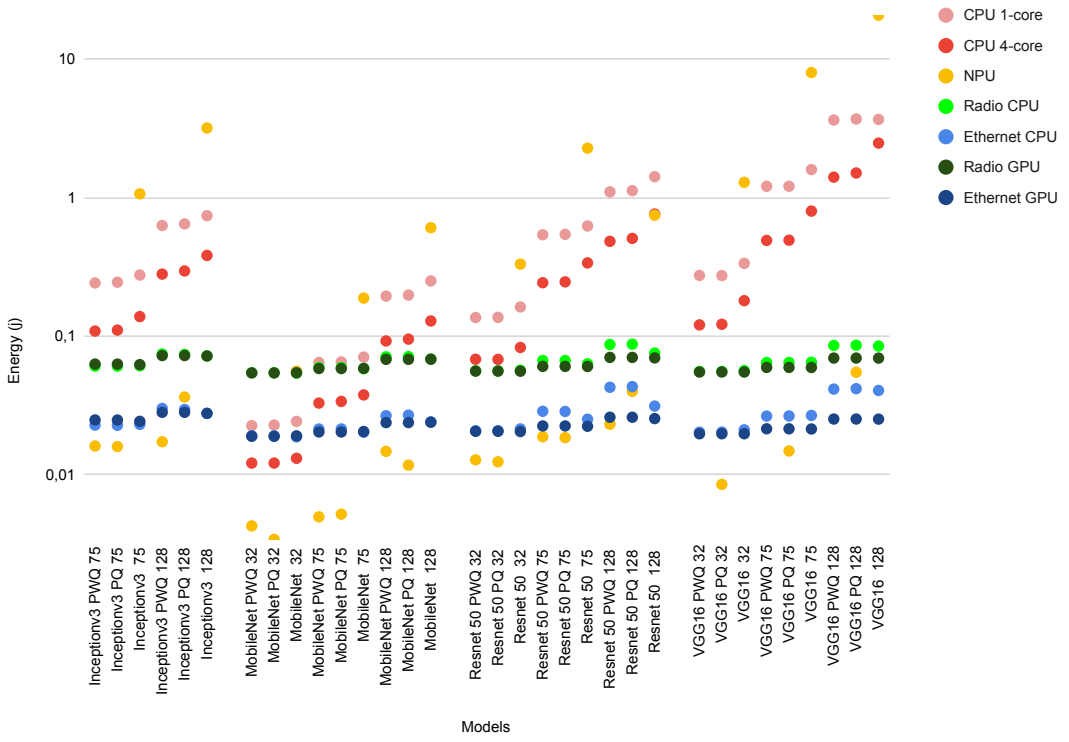


Figure 4.5: Graph of energy consumption obtained from measuring the power over the execution of the inference on different compute platforms, with lower idle energy consumption. Threshold for radio at 32 mJ and Ethernet at 23 mJ. Y-axis is logarithmic.

4.3 Energy Delay Product

The energy delay product(EDP) is a metric that provides a trade off between latency and energy consumption, as shown in equation 4.1. The EDP can be seen in Figure 4.6 and 4.7 for high and low idle power respectively.

$$\text{EDP} = \text{Energy per inference} * \text{Latency} \tag{4.1}$$

From analysing Figure 4.6 and 4.7, the same pattern as for energy and latency can be observed. This is because inference latency and energy consumption is strongly correlated which can clearly be observed in Figure 4.8.

Table 4.3: Threshold energy high idle

	Average (mJ)	stdev (mJ)	max (mJ)	min (mJ)
Radio CPU	220	35	291	180
Ethernet CPU	89	25	144	62
Radio GPU	207	21	241	180
Ethernet GPU	76	9	94	63

Table 4.4: Threshold energy low idle

	Average (mJ)	stdev (mJ)	max (mJ)	min (mJ)
Radio CPU	66	11	87	54
Ethernet CPU	27	7	43	19
Radio GPU	62	6	72	54
EThernet GPU	23	3	28	19

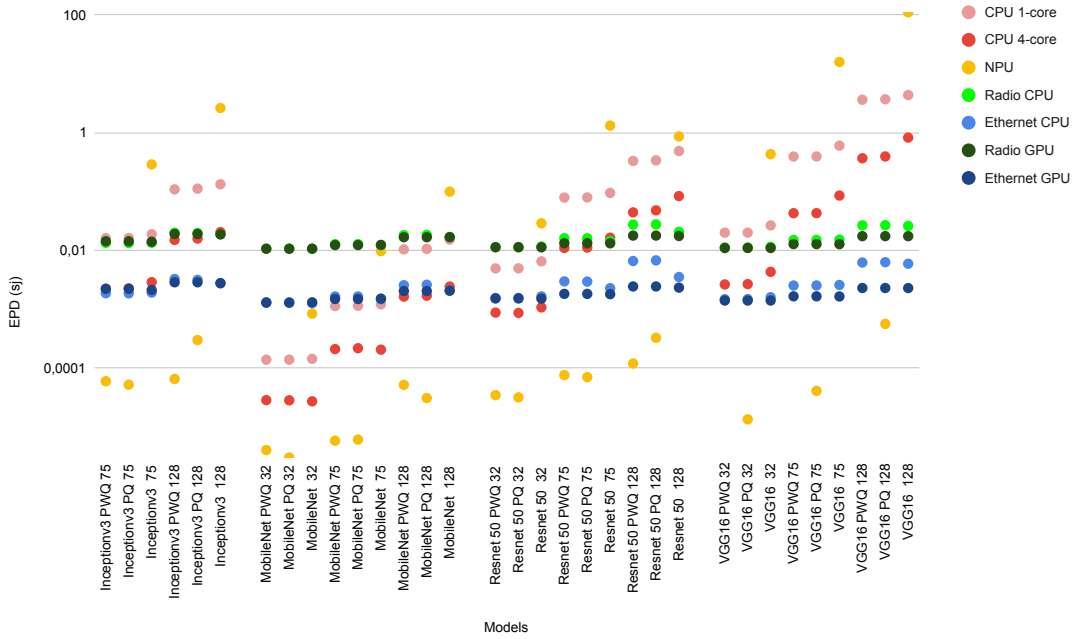


Figure 4.6: Energy delay product(EDP) for each model and compute platform with high idle.Y-axis is logarithmic.

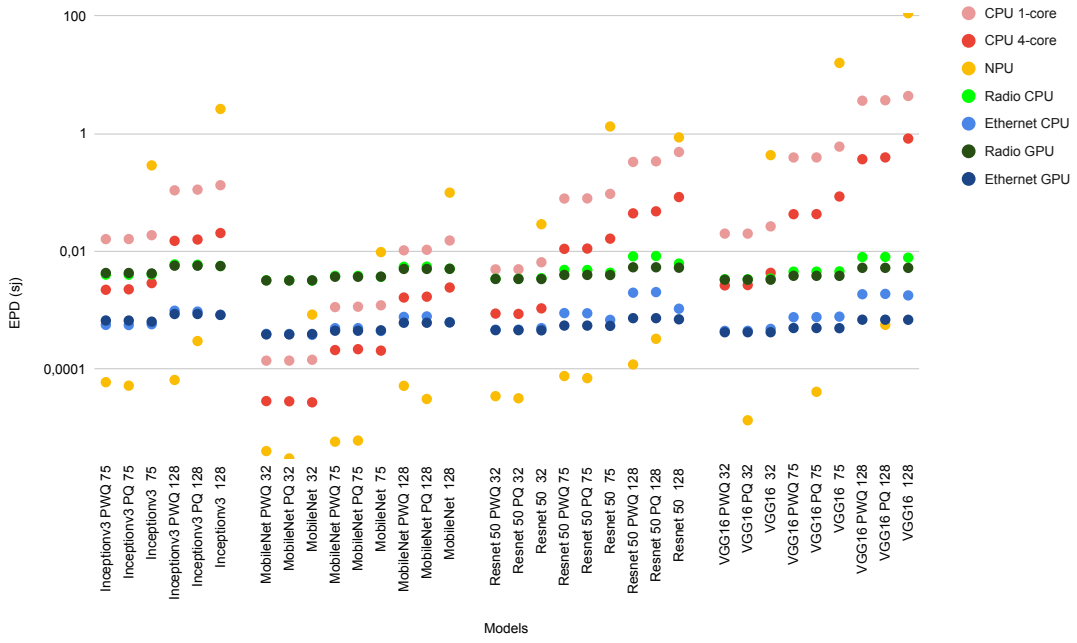


Figure 4.7: Energy delay product(EDP) for each model and compute platform with low idle.Y-axis is logarithmic.

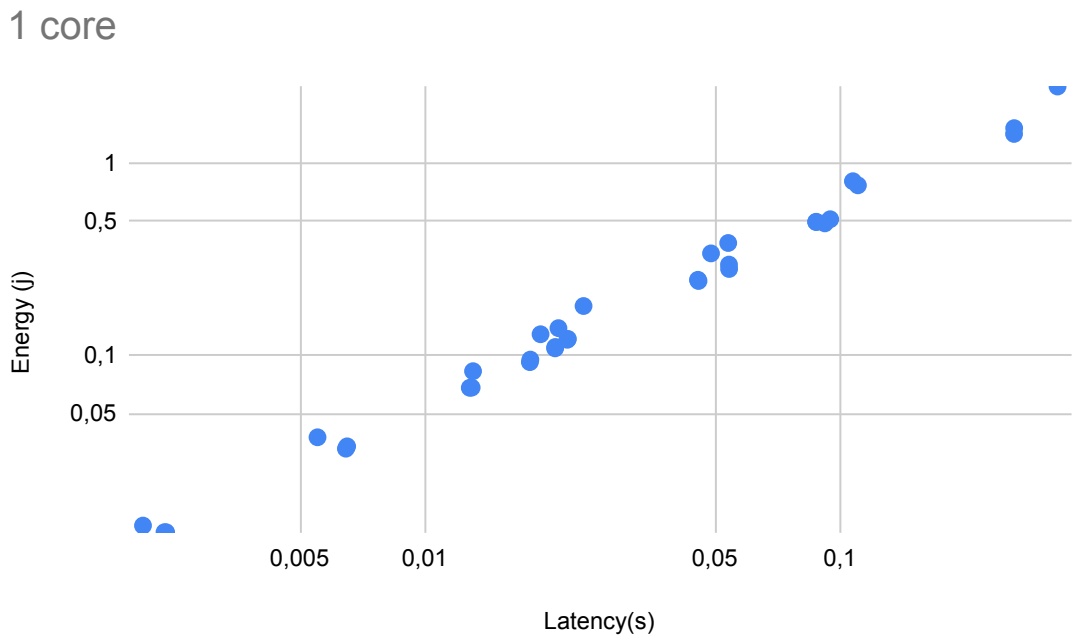


Figure 4.8: Correlation between energy and latency for 1 embedded CPU core. Both axis are logarithmic.

For the embedded platform these correlations are 99%, 98%, and 99% for 1 CPU core, 4 CPU cores, and the NPU respectively, as can be seen in Figures A.5 - A.6. This correlation is even more clear for the offloading scenarios, as seen in Figures A.7 - A.10 which is for the lower idle power case. For the offloading case the correlation is rounded up to 100% for both Ethernet and radio offloading to either CPU or GPU, for the sake of space the plots for the case featuring the high idle consumption are not included as they show the same result. This high degree of correlation is connected to how offloading is modeled in this work, as seen in equation 2.5. Equation 2.5 show that the total energy consumed from offloading is the energy used while waiting for the results and the energy used to send and receive data. In this case the energy consumed for sending and receiving data is negligible when compared to the idle energy.

5
Discussion

5.1 Neural Network Compression

Previous in chapter result 4, the latency and energy thresholds were discovered for both inference latency and energy consumption. These thresholds remained fairly consistent for all variants of a model network (PQ, PWQ and uncompressed). This is because the offloading host computer featured small variance in the inference time for the different model variations.

While compression does not seem to have any noticeable effect on performance on offloading scenarios the compression has noticeable effect on the edge device inference time. The impact of the compression is most noticeable when comparing inference latency's on the NPU. As shown in section 4.1 the average speedup for PQ models was 188x compared to uncompressed models on the NPU. The edge device CPU also featured a consistent speedup of 6-8% for both 1 core and 4 core execution, when compressing models with pruning and quantization. For CPU execution there was no noticeable performance change when compressing models with pruning and quantization (PQ) or with pruning weight clustering and quantisation (PWQ).

5.2 Model Properties

From the results described in section 4 and in appendix A there are numerous of possible discussions to be made around how different types of networks favor different compute platforms. As shown in Figures 5.1 and 5.2 the input size of the data passed to the different networks heavily affect the final result for both inference time and energy consumption, the same pattern exist for 4 cores and the npu as shown in Figures A.1-A.4.
This means that when developing a network with performance constraints it is appropriate to minimize the input size of the network as much as possible without compromising to much of the accuracy, in order to gain both a good performance while still having a useful network. While the input size may not be treated as a free factor parameter it is useful to know the effect input size has on performance metrics. For instance if given a VGG16 model that has to use a 80x80 input, then that in becomes evident that if the embedded platform does not have a NPU that offloading would be the better option when considering performance metrics such as inference latency and energy consumption.

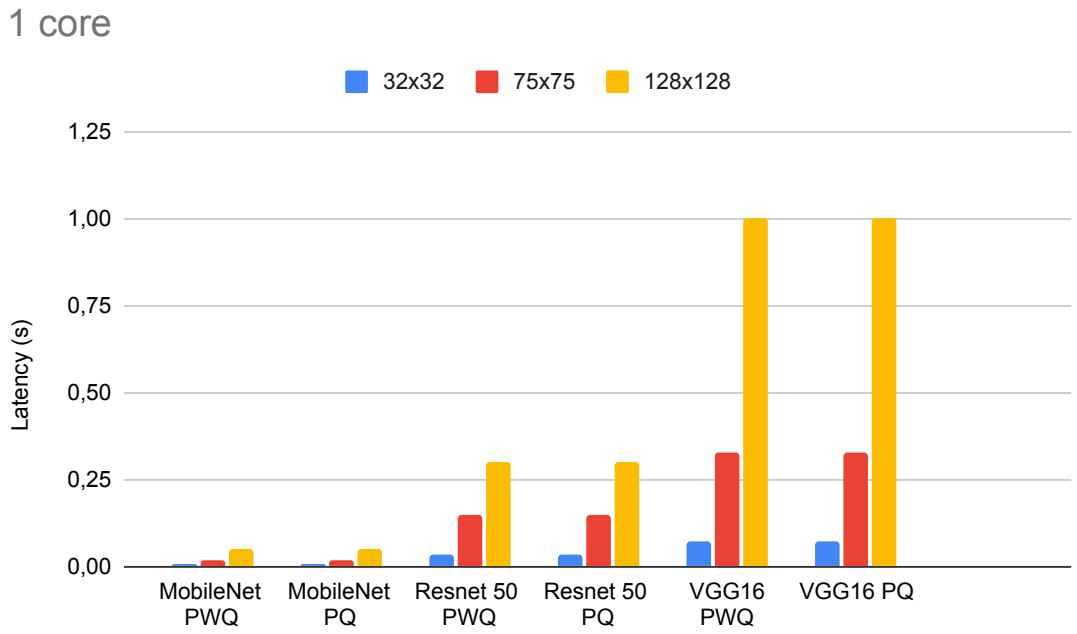


Figure 5.1: Plotting inference latency obtained from running on one CPU core on embedded device, with varying input sizes.

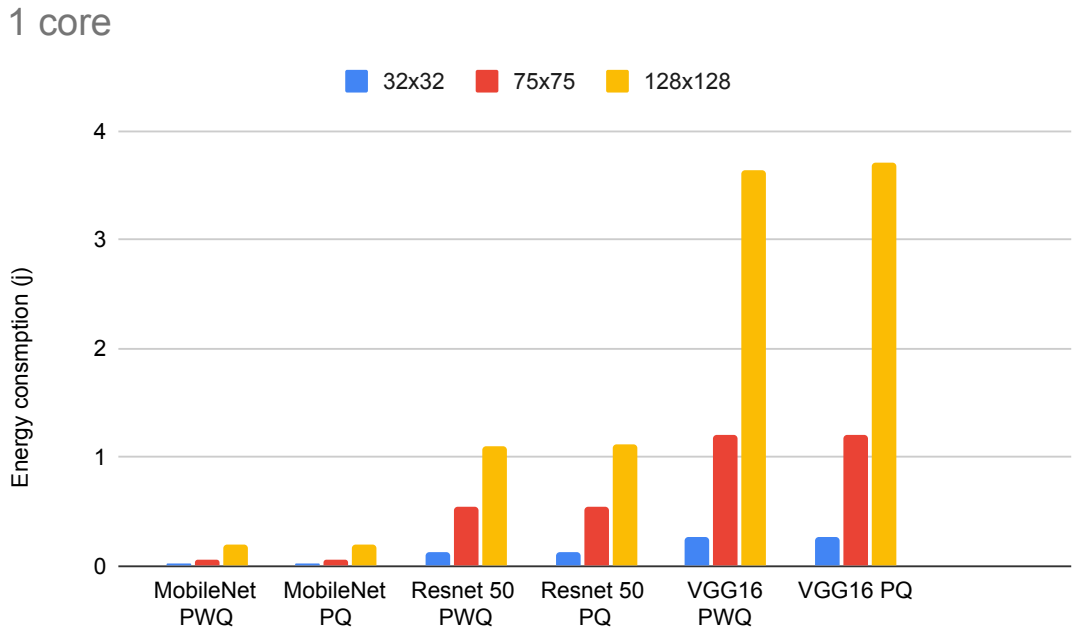


Figure 5.2: Plotting energy consumption over a single inference, obtained from running on one CPU cores on the embedded device, with varying input sizes.

We observe that the latency and the input size are closely related. Figure 5.3 describes that when the input size increase then the number of parameters grow, however we also see that the latency of the model doesn't necessarily correlate with the number of parameters. As such there must be something in the structure of the

Figure 5.3: The number parameters in networks plotted against the input size and where the size is the inference speed on one core. Shown for all full compressed networks

models that cause the difference in performance. Figure 5.4 shows how the number of parameters and depth of models effects latency.
In this Figure we see that models with fewer layers performs worse than models with more layers even when these models have more parameters. This does not necessarily mean that more layers are strictly better, results points towards that a lack of memory could be the potential cause of this problem.

As neural networks consist of a number of layers which get executed sequentially. Models that are too large to be stored in memory all at once, need to continuously load the weights of the following layers into memory during execution[45]. If the model is able to fit inside L1 cache compared to L2 cache, fewer cache collisions occur. Similar holds for models able to fit inside L2 cache compared to RAM. Fewer cache collisions results in faster execution, given that memory is being reused, as it avoids to fetch data from slower memory levels fewer times. Knowing that the CPU on the NXP card has 32kB L1 D cache, 512kB L2 cache, and 868kB RAM[25] and knowing how many parameters each layer in the models contain, it was then possible to calculate how much of the layers fit in respective memory level. Table 5.1 shows how well the average layers fits in respective memory level for each of the models. There are two major points of interests. VGG16 being the slowest network

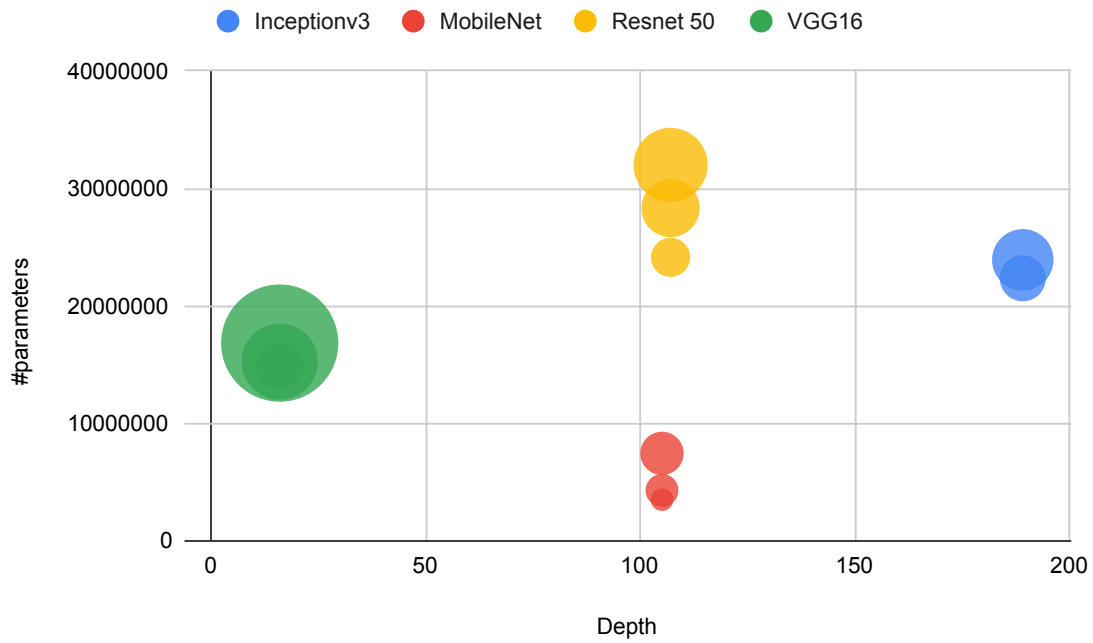


Figure 5.4: The number parameters in networks plotted against the networks depth and where the size is the inference speed on one core. Shown for all full compressed networks

has considerably the worst fit percentage out of the tested networks. Inversely we see that the fastest network has a much higher percentage. This trend can also be seen when plotting the fit percentage against inference latency for the different memory levels as seen in Figure 5.5-5.7.

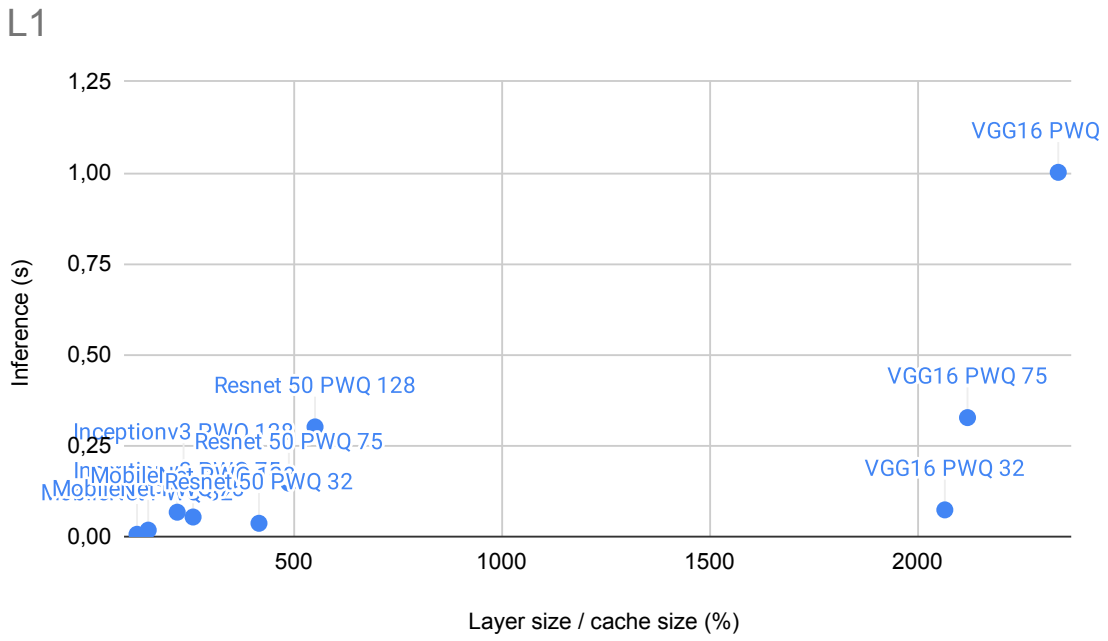


Figure 5.5: The percentage of layers that fits in L1d cache plotted against the inference latency for one core for each of the full compression.

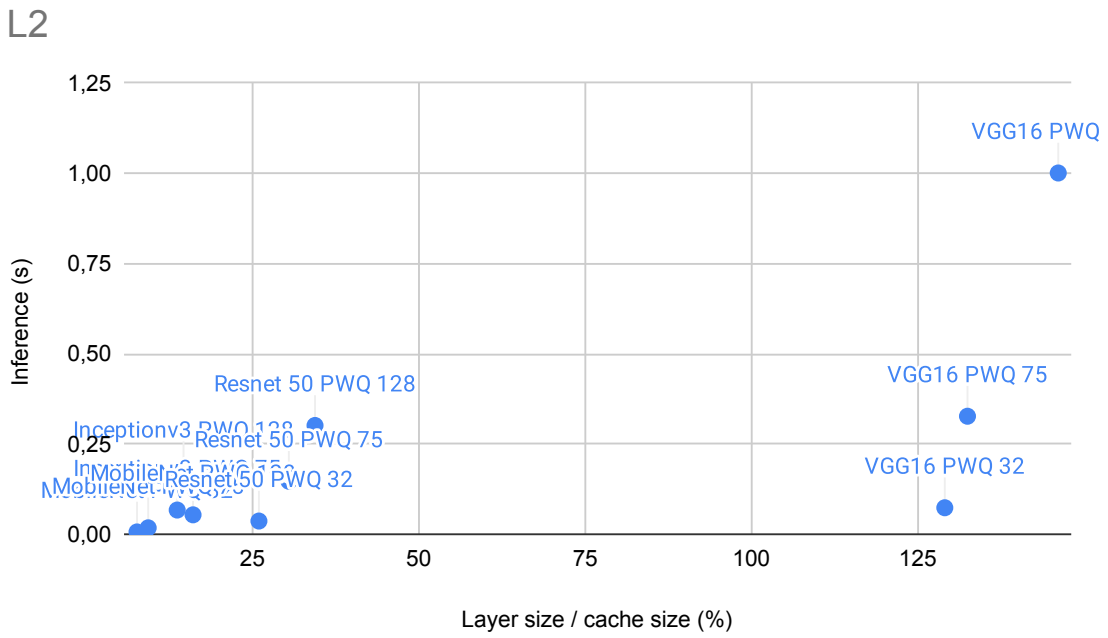


Figure 5.6: The percentage of layers that fits in L2 cache plotted against the inference latency for one core for each of the full compression.

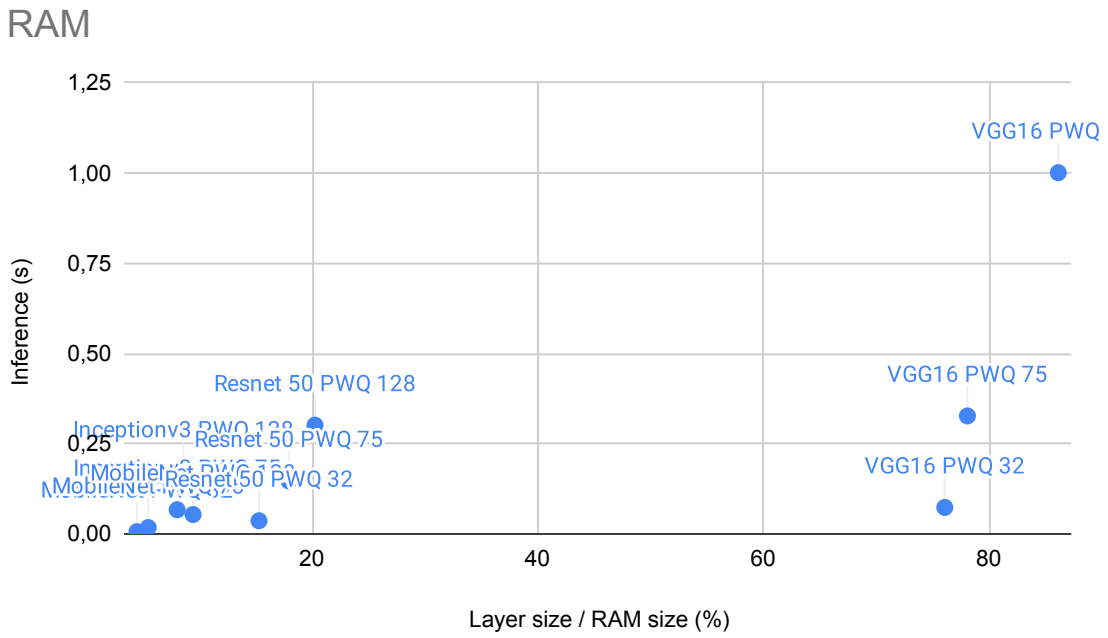


Figure 5.7: The percentage of layers that fits in RAM plotted against the inference latency for one core for each of the full compression.

This shows a correlation between inference latency and the number of parameters in each layer and the sizes of device memory structures.

The same trend can be seen when looking at the results from perf. It can be observed that the number of instructions and L1-dcache-loads executed correlate well with the performance latency for one core, while relation is somewhat less clear for four cores which might be due to parallelism factors, see Figures 5.8 & A.17 as well as Figures 5.9 & A.19.
This was expected as was when the size of the data to process increase, then the number of instructions needed to execute increase, and as such the time need to execute does. As such we would expect the number of instructions and data loads to correlated against the number of parameters, however this is not what can be seen in the Figures A.26-A.31. While the number of L1d cache misses are fairly similarly for 1 and 4 cores and correlates well against the latency. Figures 5.10 and A.21.
None of these correlation was observed for the NPU, this might be due to how the inference is being accelerated as can be seen in Figures A.18, A.20 and A.22.

5.3 Thresholds

From Figure 4.3 - 4.2 it becomes clear that communication is the dominant factor and as progress is made in the communication sector the threshold for when to offload is likely to shift in favour of offloading. A clear case of this is the upcoming 5G technology, which promises reliable (User plane) latency of 1-4 milliseconds

Table 5.1: Average layer fit for respective model and for each memory level

Model name	L1 (%)	L2(%)	RAM(%)
Inceptionv3 75	217	14	8
Inceptionv3 128	233	15	9
Resnet50 32	414	26	15
Resnet50 75	486	30	18
Resnet50 128	549	34	20
Mobilenet 32	120	8	4
Mobilenet 75	148	9	5
Mobilenet 128	255	16	9
VGG16 32	2063	129	76
VGG16 75	2117	132	78
VGG16 128	2336	146	86

depending on usage scenario[46].

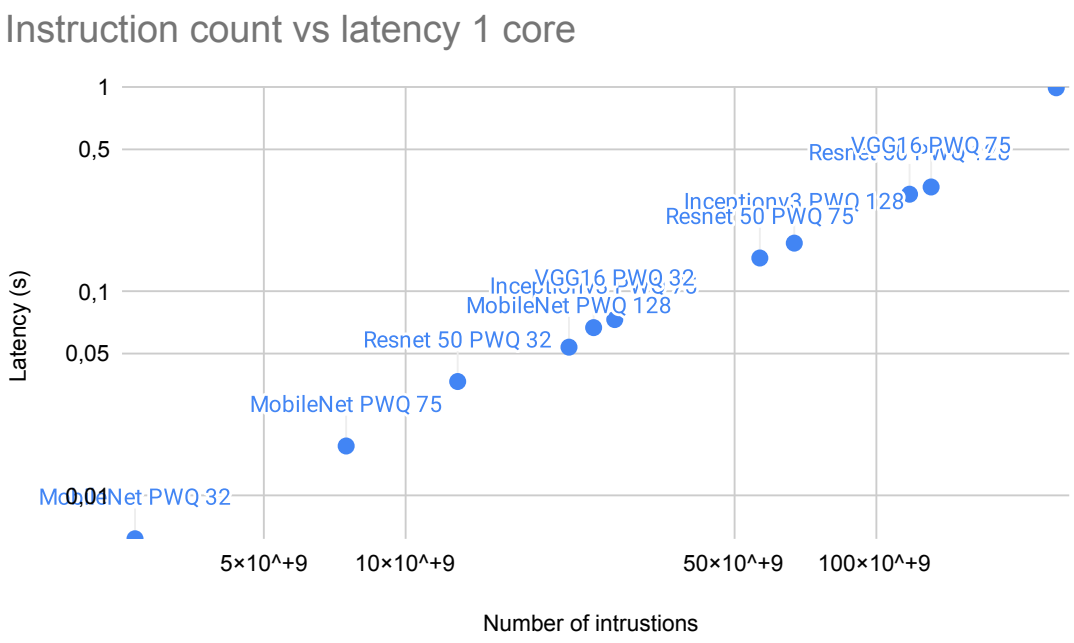


Figure 5.8: Number of instructions plotted against the latency on 1 core for receptive model. Both axis are logarithmic

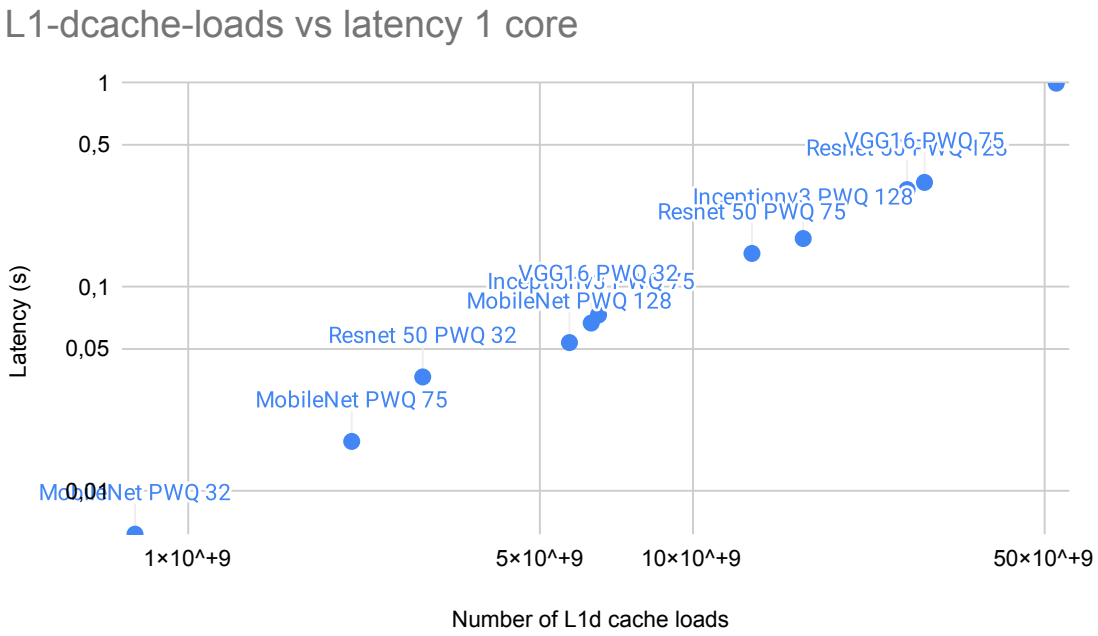


Figure 5.9: Number of l1d cache loads plotted against the latency on 1 core for receptive model. Both axis are logarithmic

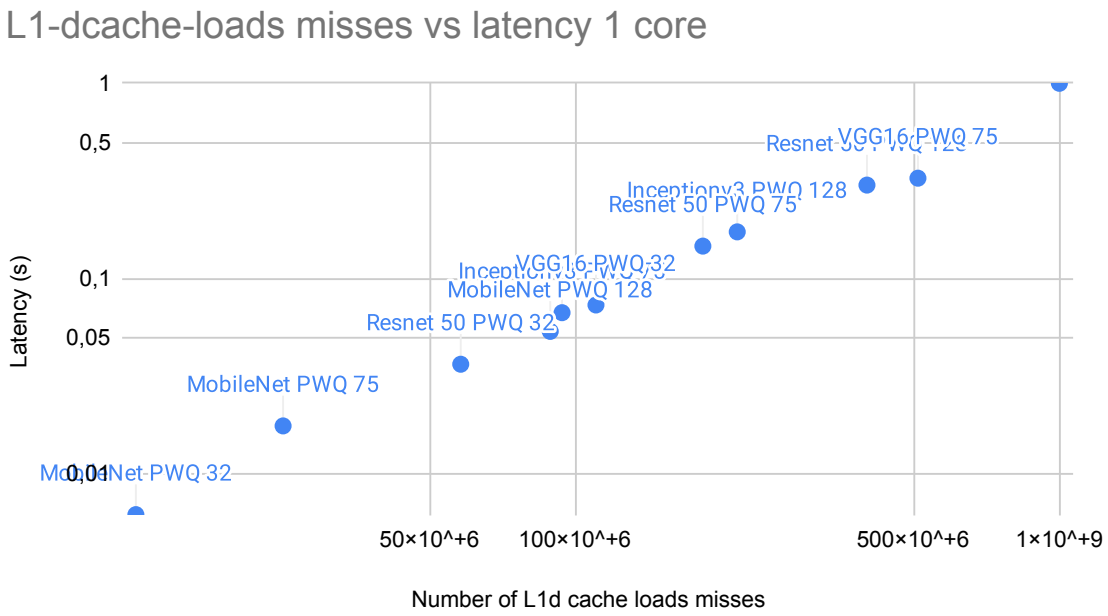


Figure 5.10: Number of l1d cache loads misses plotted against the latency on 1 core for receptive model. Both axis are logarithmic

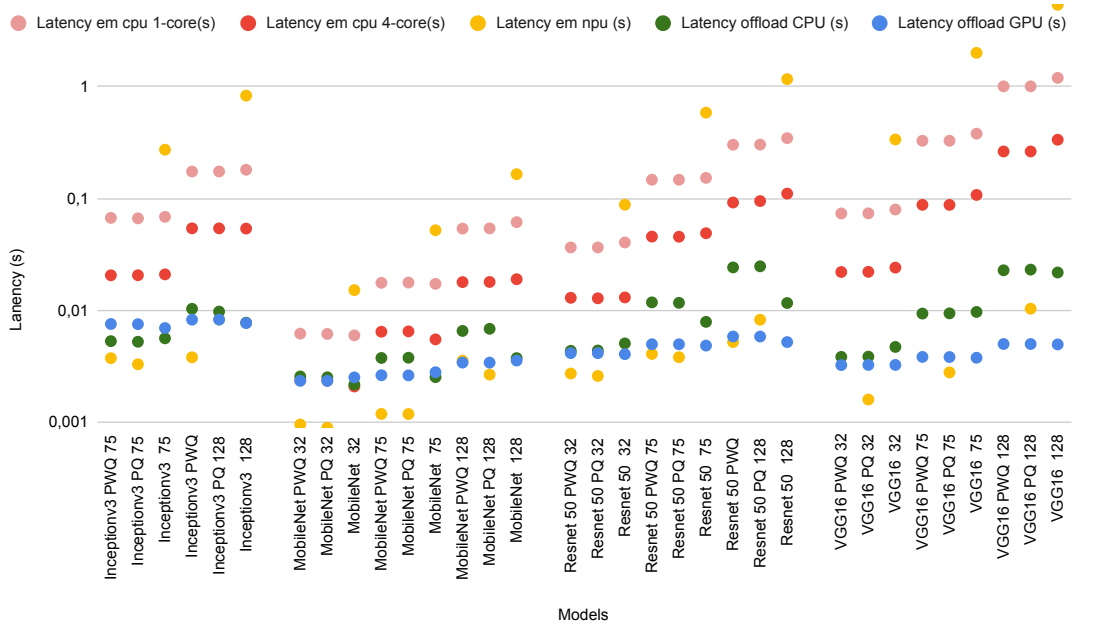


Figure 5.11: Illustration of latency contributions of network communication and execution time of model, using 5G to offload to CPU

User plane latency is defined as the one-way time it takes to deliver an application layer message from the radio protocol layer. If it is possible to offload models to the base stations then this would result in roughly a 6 times speedup. The progression of communication technology does not end with 5G, the development 6G is underway [47]. With further improvements on communication latency offloading will likely become more competitive as time goes on, given that embedded CPUs and accelerators improve at a similar rate.

When modeling same offloading previously shown but with communication parameters from 5G, that being 1 ms latency and 202 Mbps [48]. This considered changes the offloading breakout as seen in Figures 5.12 and A.13 .

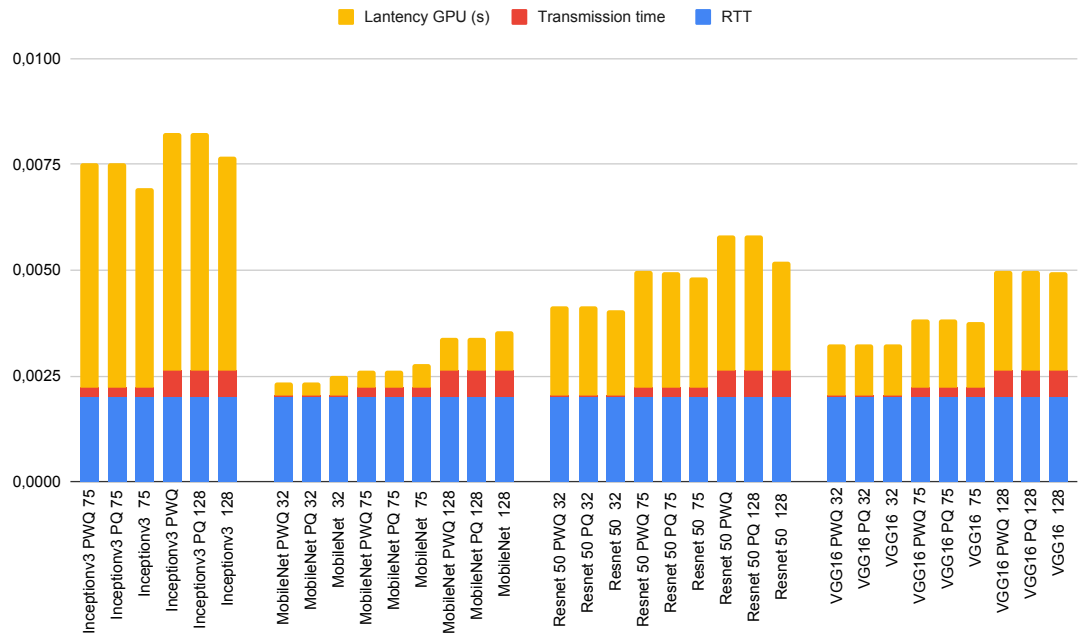


Figure 5.12: Illustration of latency contributions of network communication and execution time of model, using 5G to offload to GPU

This show that for 5G offloading the inference latency would be the dominant part, this is contrary to what can be seen for radio and Ethernet offloading, as seen in Figures 4.2, 4.3, A.11, and A.12. This improvement in offloading latency also has a considerable impact on the offloading decision. As can be seen in Figure 5.11, offloading out performs the inference on CPU for all model expect Mobilenet 32 and even out performs the NPU for some models.

Noteworthy is that it was found that the lowest in this work obtainable idle energy consumption of the *i.MX 8M Plus* was 1 watt. This is comparatively high compared to other embedded devices such as for example *Raspberry Pi Zero 2 W* [49, 50] that has an obtainable idle consumption of 600 mW. This means that other devices may compromise execution power for a lower power consumption that will result in a different outcome in regards to latency and energy consumption and as well EDP.

5.4 Other observations

From analysing the results we can also conclude that Mobilenet is well suited for embedded devices. This is seen as its on-chip CPU and NPU inference latency is the lowest among all tested networks. Together with its low energy consumption it outperforms offloading in all scenarios with the exception of the uncompressed variant. This was expected since Mobilenet is designed with the core intention of achieving good performance on embedded devices, and our experiment validates that intention.

It could also be relevant to briefly discuss the accuracy of the evaluated models, for this propose we will use Keras achieved accuracy on Mobilenet[51]. Note that input size used in Keras evaluation is 224x224 which is larger than the input sizes considered in this study, as such the result are not directly comparable but give a relative accuracy for the different models. Keras results show that both Mobilenet(v2) and VGG16 achieved an accuracy 90.1%. When combined with the result of this study, Mobilenet becomes the obvious choice as it has considerable better performance. While Resnet50 and InceptionV3 achieved an accuracy of 92.1% and 93.7% respectively, however these models are larger and slower when compared to Mobilenet, as such there is a accuracy performance trade off to keep in mind.

5.5 Future Works

As this work only consider the energy consumption on the embedded device and not the entire system, a complementary study looking at the energy used for the entire system would give a better holistic understanding of benefits and cons of offloading. Such studies could have a greater focus on network energy consumption, such as base station and routers as well as comparing dedicated servers compared with cloud computing solutions. This would better the understanding on how the decision of where to offload and how the communication medium used to offload affect the environmental factors and infrastructure costs.

In this work only the native offloading case was considered, it would be interesting to see how more advanced offloading techniques such as *PerDNN*[24]. This would give insight how energy efficient modern offloading techniques are and such better map the solution space.

While this work is not considering model accuracy when evaluating the different model architectures, the *Deep compression*[12] paper shows that high compression can be achieved with little or zero accuracy loss. With that it might be valuable to perform additional studies where accuracy is considered since it can potentially yield additional insight to the decision of offloading or on-chip computation. This is because a hard accuracy constraint might exclude some model architectures since smaller is not always better in that sense.

In this work we have shown some correlations between neural network designs and the latency and energy efficiency, however a study focused on how network design affect performance could be useful for future embedded AI research. Such study could explore how to optimize a model for the memory structure and how such optimizations effect accuracy and performance.

6
Conclusion

The data visualised in Figures 4.1 and 4.4 indicates a strong correlation with NPU performance and model compression techniques. This can be seen since all uncompressed models perform significantly worse on the embedded NPU than compressed models. Compressed models obtains an average 188x speedup compared to non compressed models when executed on the NPU. With this we conclude that it is of best interest to compress neural networks using compression techniques such as quantization, pruning when deploying neural network models on the *i.MX 8M Plus* NPU in order to gain the best performance possible. In addition it is said that utilizing weight clustering can be used in the case where hardware and software support is available, but does not contribute to other performance gains otherwise.

We conclude that when the inference time aspect is of the highest priority it is for all our tested networks best to perform the calculation on the edge device's NPU if one is available. This conclusion is made based on the fact that the NPU had the fastest inference time for all tested compressed networks when including the communication for the offloading scenario. On the other hand the answer is not as clear when it comes to energy consumption since if the idle power can be lowered while waiting for response from host the result is not unanimous.

As been shown in this work the offloading performance is heavily influenced by the round trip time, and as such the decision whether to offload or not in regards to performance is heavily dependent on the commutation media used. Using the global average latencies for wired and wireless commutation [41], and the offloading platform use in this work, then thresholds for single inferences are around 24 ms and 68 ms for Ethernet and radio offloading respectively. While for newer commutation techniques or when the compute server moves closer to the edge device then the inference time on the offloading becomes less influential on the total latency. With data from *Minimum requirements related to technical performance for IMT-2020 radio interface (s)* [46] newer 5G communication would be able to obtain network latencies of around 1-4 ms making offloading a much more competitive option, assuming hardware performance of devices tested in this work.

Bibliography

[1] Diaa Salama AbdELminaam et al. "A deep facial recognition system using computational intelligent algorithms". In: *Plos one* 15.12 (2020), e0242269.

[2] Mrinal R Bachute and Javed M Subhedar. "Autonomous Driving Architectures: Insights of Machine Learning and Deep Learning Algorithms". In: *Machine Learning with Applications* 6 (2021), p. 100164.

[3] Nitthilan Kannappan Jayakodi et al. "Trading-Off Accuracy and Energy of Deep Inference on Embedded Systems: A Co-Design Approach". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2881–2893. DOI: 10.1109/TCAD.2018.2857338.

[4] Nasir Abbas et al. "Mobile Edge Computing: A Survey". In: *IEEE Internet of Things Journal* 5.1 (2018), pp. 450–465. DOI: 10.1109/JIOT.2017.2750180.

[5] Sam Leroux et al. *Privacy Aware Offloading of Deep Neural Networks*. 2018. arXiv: 1805.12024 [cs.LG].

[6] Shuochao Yao et al. "Deep Compressive Offloading: Speeding up Neural Network Inference by Trading Edge Computation for Network Latency". In: *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 476–488. ISBN: 9781450375900. URL: https://doi.org/10.1145/3384419.3430898.

[7] Sun-Chong Wang. "Artificial Neural Network". In: *Interdisciplinary Computing in Java Programming*. Boston, MA: Springer US, 2003, pp. 81–100. ISBN: 978-1-4615-0377-4. DOI: 10.1007/978-1-4615-0377-4_5. URL: https://doi.org/10.1007/978-1-4615-0377-4_5.

[8] Shih-Chung B. Lo et al. "Artificial convolution neural network for medical image pattern recognition". In: *Neural Networks* 8.7 (1995). Automatic Target Recognition, pp. 1201–1214. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(95)00061-5. URL: https://www.sciencedirect.com/science/article/pii/0893608095000615.

[9] Cesare Alippi, Simone Disabato, and Manuel Roveri. "Moving Convolutional Neural Networks to Embedded Systems: The AlexNet and VGG-16 Case". In: *2018 17th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 2018, pp. 212–223. DOI: 10.1109/IPSN.2018.00049.

[10] Subarna Tripathi et al. "LCDet: Low-Complexity Fully-Convolutional Neural Networks for Object Detection in Embedded Systems". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2017, pp. 411–420. DOI: 10.1109/CVPRW.2017.56.

[11] Alfredo Canziani, Eugenio Culurciello, and Adam Paszke. "Evaluation of neural network architectures for embedded systems". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2017, pp. 1–4.

[12] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV].

[13] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[14] Lu Hou and James T Kwok. "Loss-aware weight quantization of deep networks". In: *arXiv preprint arXiv:1802.08635* (2018).

[15] Yann LeCun, John Denker, and Sara Solla. "Optimal brain damage". In: *Advances in neural information processing systems* 2 (1989).

[16] Stephen Hanson and Lorien Pratt. "Comparing biases for minimal network construction with back-propagation". In: *Advances in neural information processing systems* 1 (1988).

[17] Babak Hassibi and David Stork. "Second order derivatives for network pruning: Optimal brain surgeon". In: *Advances in neural information processing systems* 5 (1992).

[18] Nikko Ström. "Phoneme probability estimation with dynamic sparsely connected artificial neural networks". In: *The Free Speech Journal* 5.1-41 (1997), p. 2.

[19] IBM Cloud Education. *What is overfitting?* Mar. 2021. URL: https://www.ibm.com/cloud/learn/overfitting.

[20] *Tensorflow model optimization toolkit - weight clustering API*. URL: https://blog.tensorflow.org/2020/08/tensorflow-model-optimization-toolkit-weight-clustering-api.html.

[21] *Weight clustering: tensorflow model optimization*. URL: https://www.tensorflow.org/model_optimization/guide/clustering.

[22] Tianshi Chen et al. "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 269–284. ISBN: 9781450323055. DOI: 10.1145/2541940.2541967. URL: https://doi.org/10.1145/2541940.2541967.

[23] Reza Hojabr et al. "SkippyNN: An Embedded Stochastic-Computing Accelerator for Convolutional Neural Networks". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[24] Hyuk-Jin Jeong et al. "PerDNN: Offloading Deep Neural Network Computations to Pervasive Edge Servers". In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 1055–1066. DOI: 10.1109/ICDCS47774.2020.00114.

[25] *i.MX 8M Plus Applications Processor Datasheet for Industrial Products*. IMX8MPIEC. Rev. 1. NXP Semiconductors. Aug. 2021.

[26] *Yocto project*. https://www.yoctoproject.org/. Accessed: 2022-02-07.

[27] Martín Abadi et al. *TensorFlow, Large-scale machine learning on heterogeneous systems*. Nov. 2015. DOI: 10.5281/zenodo.4724125.

[28] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[29] Huan Liang, Wenlong Fu, and Fengji Yi. "A Survey of Recent Advances in Transfer Learning". In: *2019 IEEE 19th International Conference on Communication Technology (ICCT)*. 2019, pp. 1516–1523. DOI: 10.1109/ICCT46805.2019.8947072.

[30] Chuanqi Tan et al. "A Survey on Deep Transfer Learning". In: *Artificial Neural Networks and Machine Learning – ICANN 2018*. Ed. by Věra Kůrková et al. Cham: Springer International Publishing, 2018, pp. 270–279. ISBN: 978-3-030-01424-7.

[31] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[32] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[33] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *CoRR* abs/1512.00567 (2015). arXiv: 1512.00567. URL: http://arxiv.org/abs/1512.00567.

[34] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[35] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[36] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[37] Vinay Uday Prabhu and Abeba Birhane. *Large image datasets: A pyrrhic win for computer vision?* 2020. DOI: 10.48550/ARXIV.2006.16923. URL: https://arxiv.org/abs/2006.16923.

[38] *Post-training quantization;Tensorflow Lite*. URL: https://www.tensorflow.org/lite/performance/post_training_quantization.

[39] Martıén Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[40] *Tfmot.sparsity.keras.prune_low_magnitude: tensorflow model optimization*. URL: https://www.tensorflow.org/model_optimization/api_docs/python/tfmot/sparsity/keras/prune_low_magnitude.

[41] *Internet speed around the world*. URL: https://www.speedtest.net/global-index.

[42] *Performance measurement*. https://www.tensorflow.org/lite/performance/measurement. Accessed: 2022-02-07.

[43] *Joulescope*. URL: https://www.joulescope.com/.

[44] Linus Torvalds. *Linux perf source code*. 2022. URL: https://github.com/torvalds/linux/tree/master/tools/perf.

[45] Jamie Hanlon. *Why is so much memory needed for deep neural networks?* Jan. 2017. URL: https://www.graphcore.ai/posts/why-is-so-much-memory-needed-for-deep-neural-networks.

[46] M Series. "Minimum requirements related to technical performance for IMT-2020 radio interface (s)". In: *Report* (2017), pp. 2410–.

[47] Anutusha Dogra, Rakesh Kumar Jha, and Shubha Jain. "A Survey on Beyond 5G Network With the Advent of 6G: Architecture and Emerging Technologies". In: *IEEE Access* 9 (2021), pp. 67512–67547. DOI: 10.1109/ACCESS.2020.3031234.

[48] Darijo Raca et al. "Beyond Throughput, the next Generation: A 5G Dataset with Channel and Context Metrics". In: *Proceedings of the 11th ACM Multimedia Systems Conference*. MMSys '20. Istanbul, Turkey: Association for Computing Machinery, 2020, pp. 303–308. ISBN: 9781450368452. DOI: 10.1145/3339825.3394938. URL: https://doi.org/10.1145/3339825.3394938.

[49] Raspberry Pi. *Raspberry Pi Zero 2 W*. URL: https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/.

[50] Jean Luc Aufranc. *A deep dive into Raspberry Pi Zero 2 W's power consumption - CNX software*. Dec. 2021. URL: https://www.cnx-software.com/2021/12/09/raspberry-pi-zero-2-w-power-consumption/.

[51] Keras Team. *Keras Documentation: Keras applications*. URL: https://keras.io/api/applications/.

A

Figures

A.1 Model input size

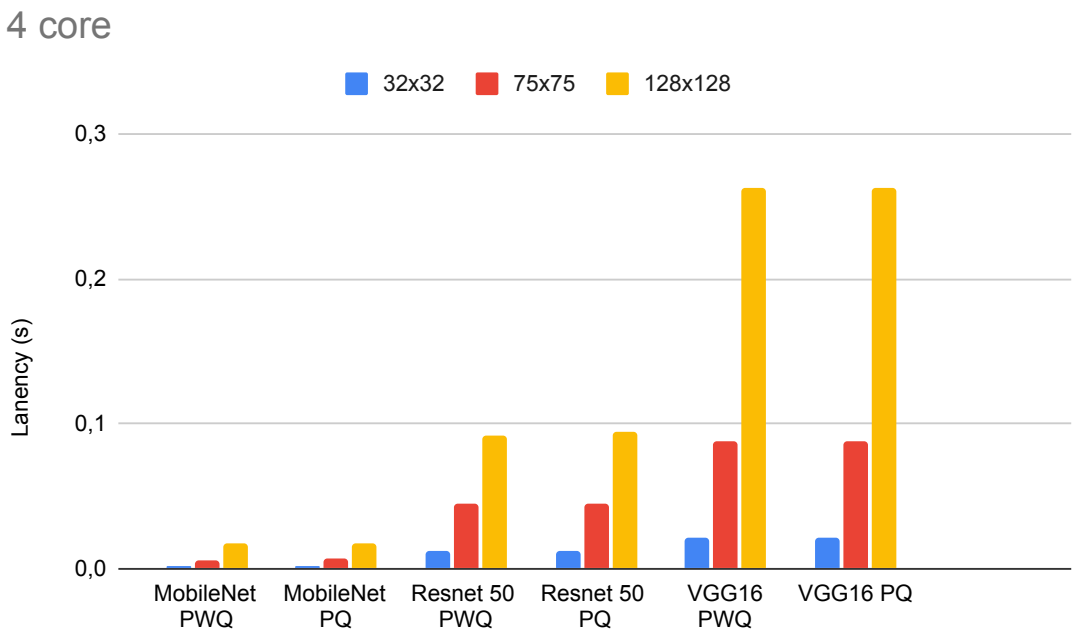


Figure A.1: Plotting inference latency obtained from running on four CPU core on embedded device, with varying input sizes.

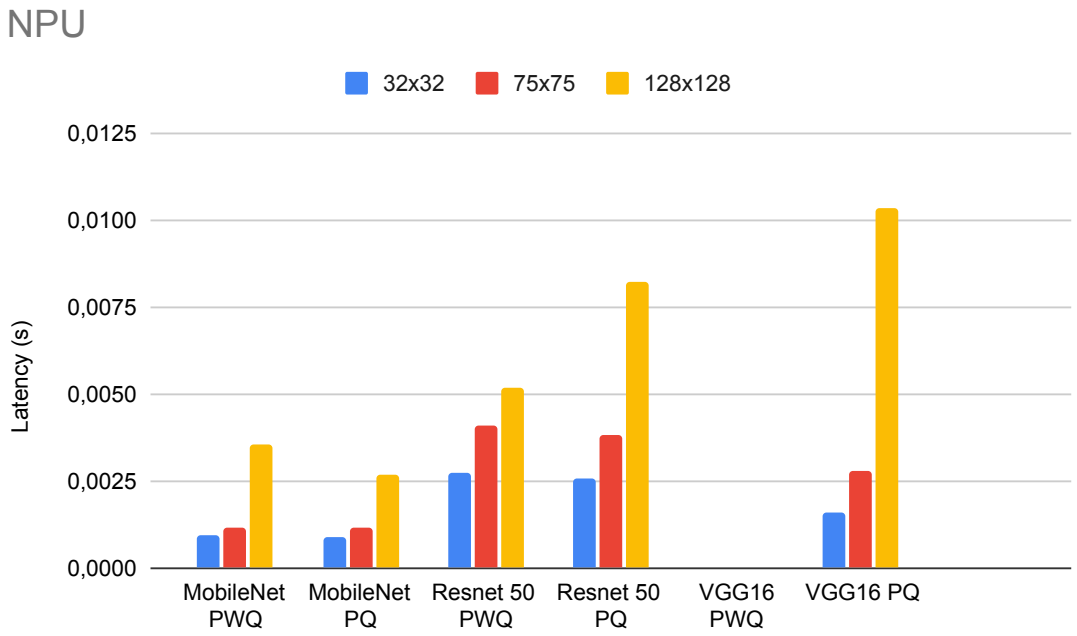


Figure A.2: Plotting inference latency obtained from running on the NPU of the embedded device, with varying input sizes.

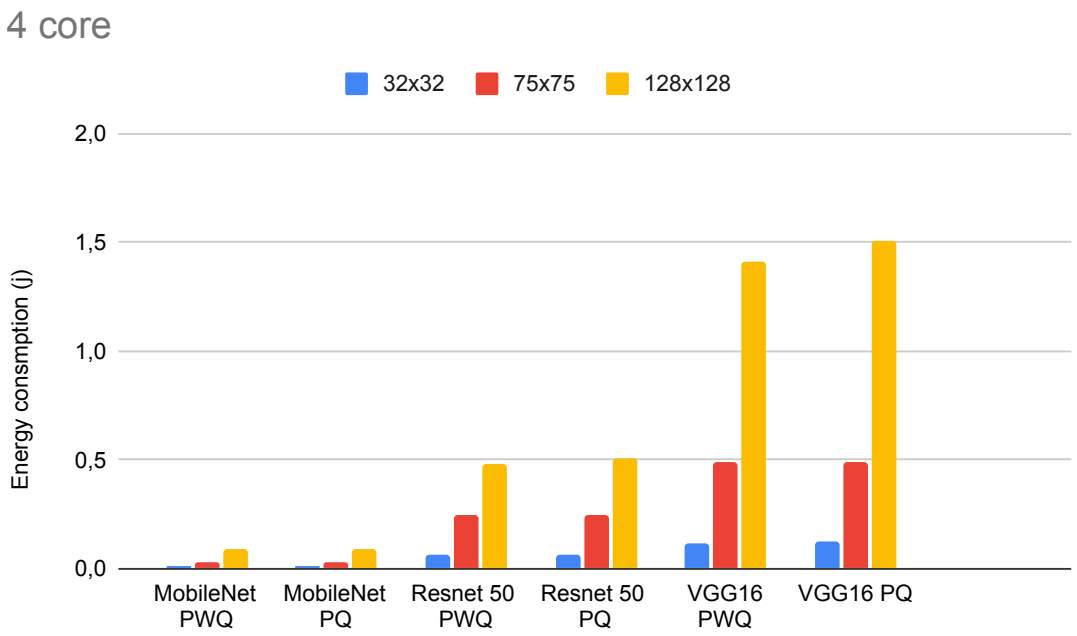


Figure A.3: Plotting energy consumption over a single inference, obtained from running on four CPU cores on the embedded device, with varying input sizes.

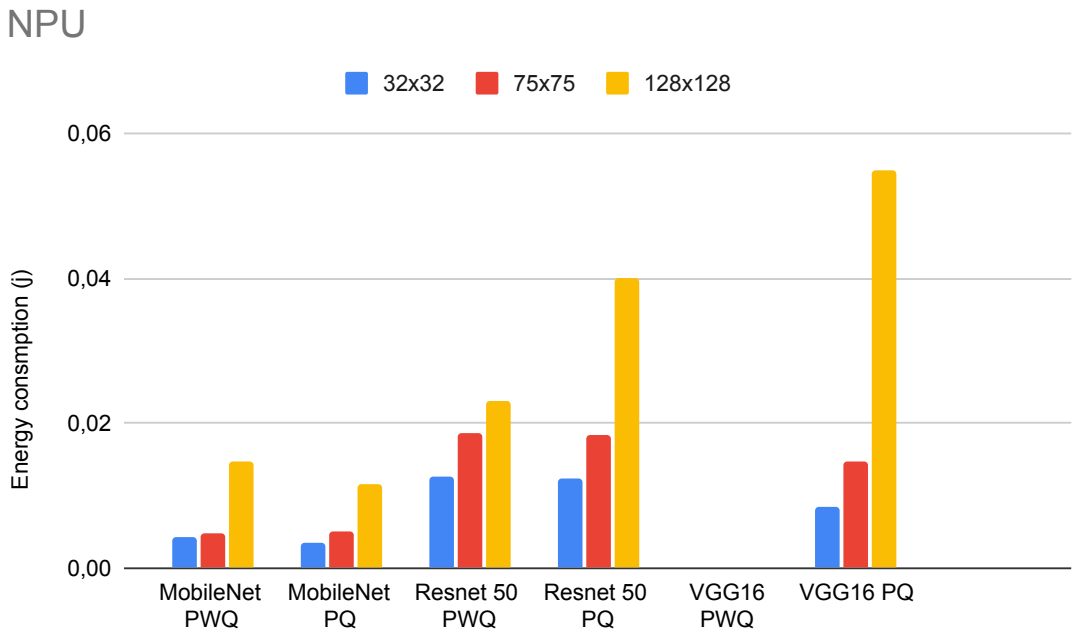


Figure A.4: Plotting energy consumption over a single inference, obtained from running on the NPU of the embedded device, with varying input sizes.

A.2 Energy Latency Correlation

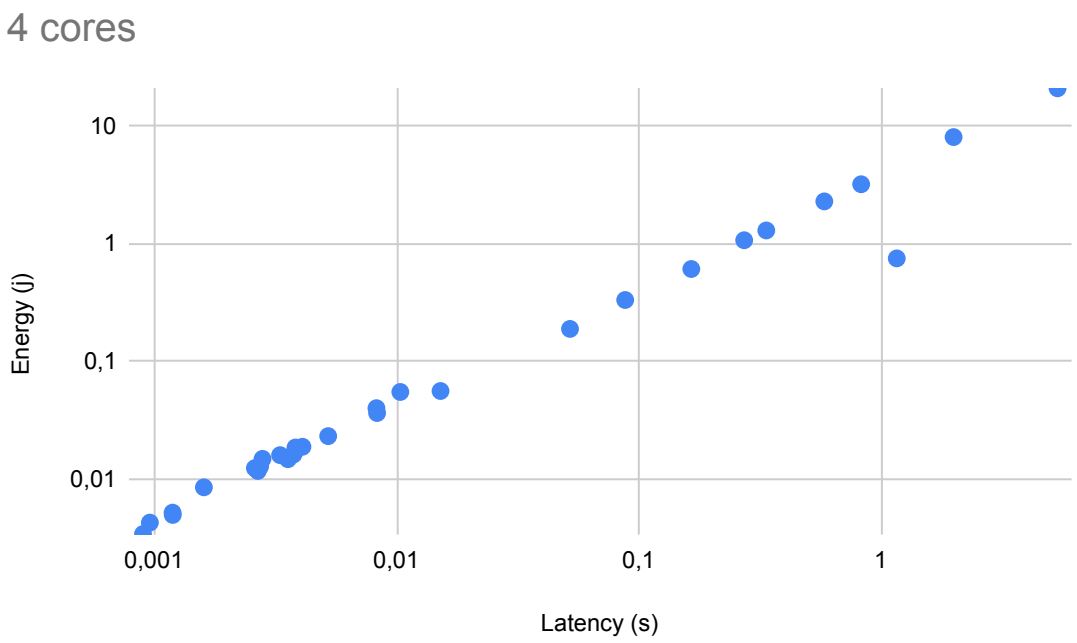


Figure A.5: Correlation between energy and latency for 4 embedded CPU core. Both axis are logarithmic.

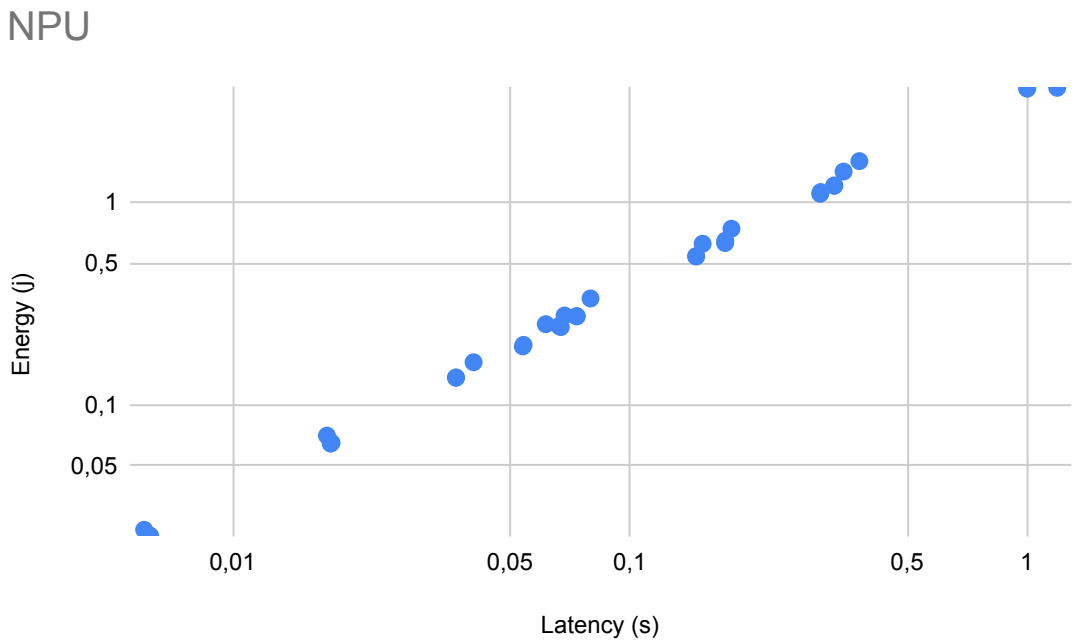


Figure A.6: Correlation between energy and latency for the embedded NPU. Both axis are logarithmic.

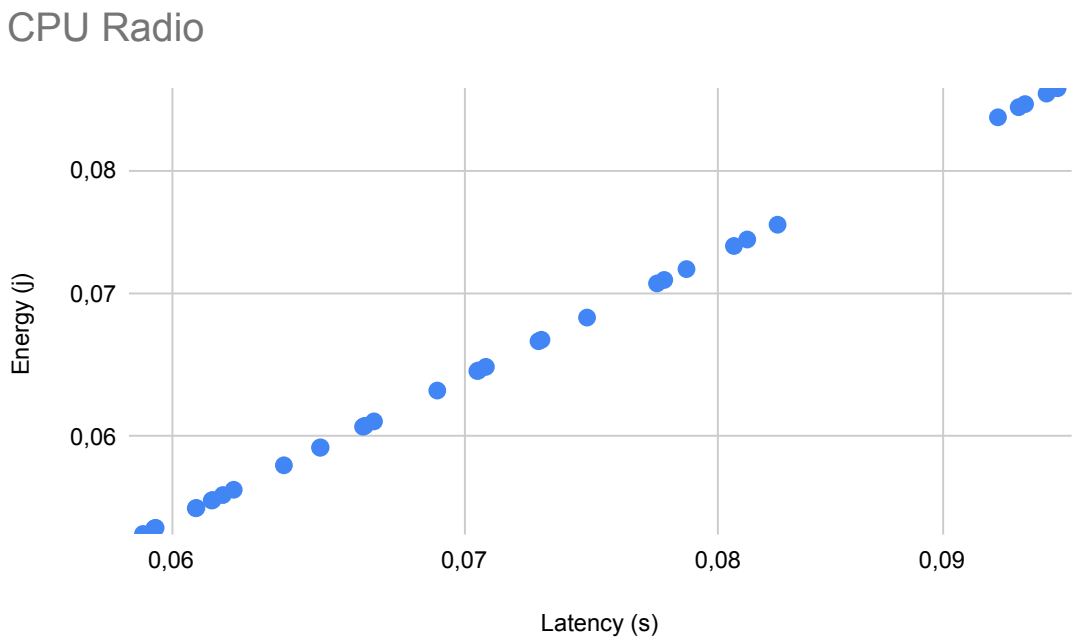


Figure A.7: Correlation between energy and latency for the embedded NPU. Both axis are logarithmic.

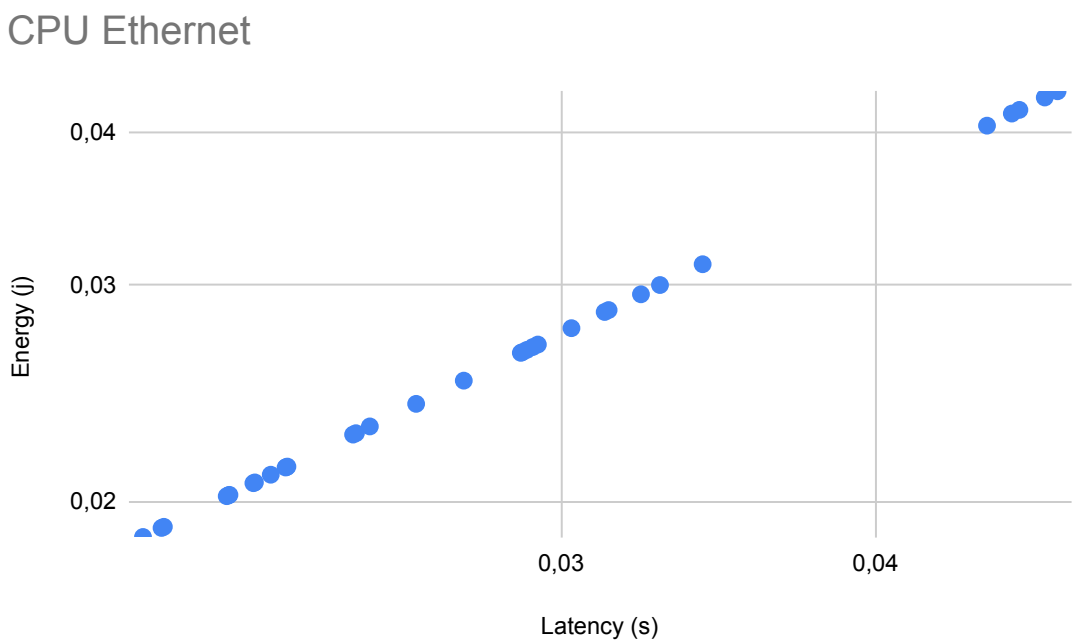


Figure A.8: Correlation between energy and latency for the embedded NPU. Both axis are logarithmic.

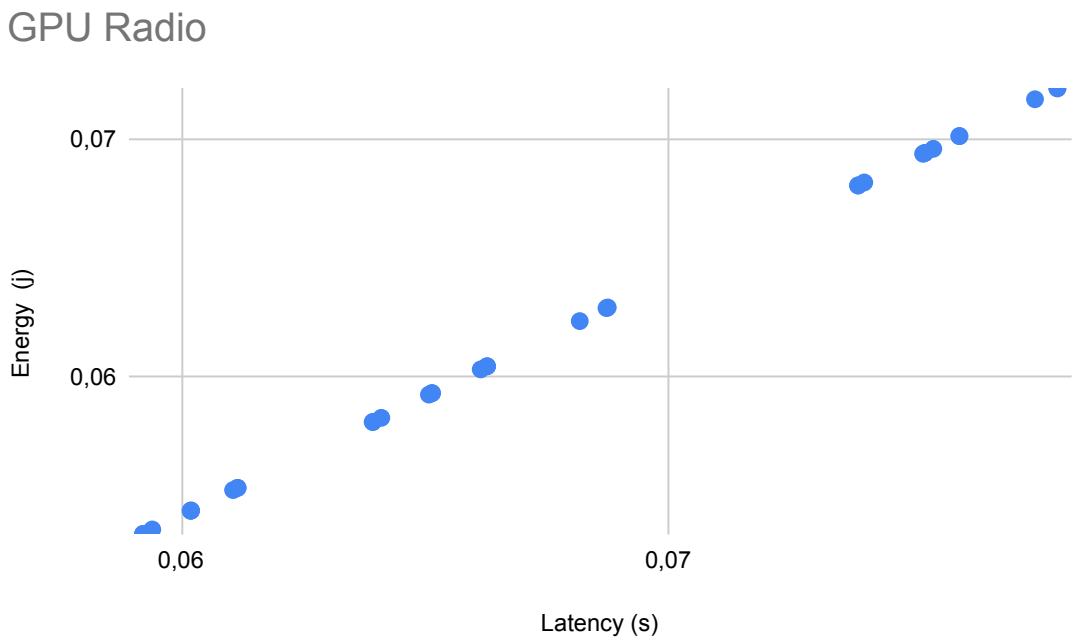


Figure A.9: Correlation between energy and latency for the embedded NPU. Both axis are logarithmic.

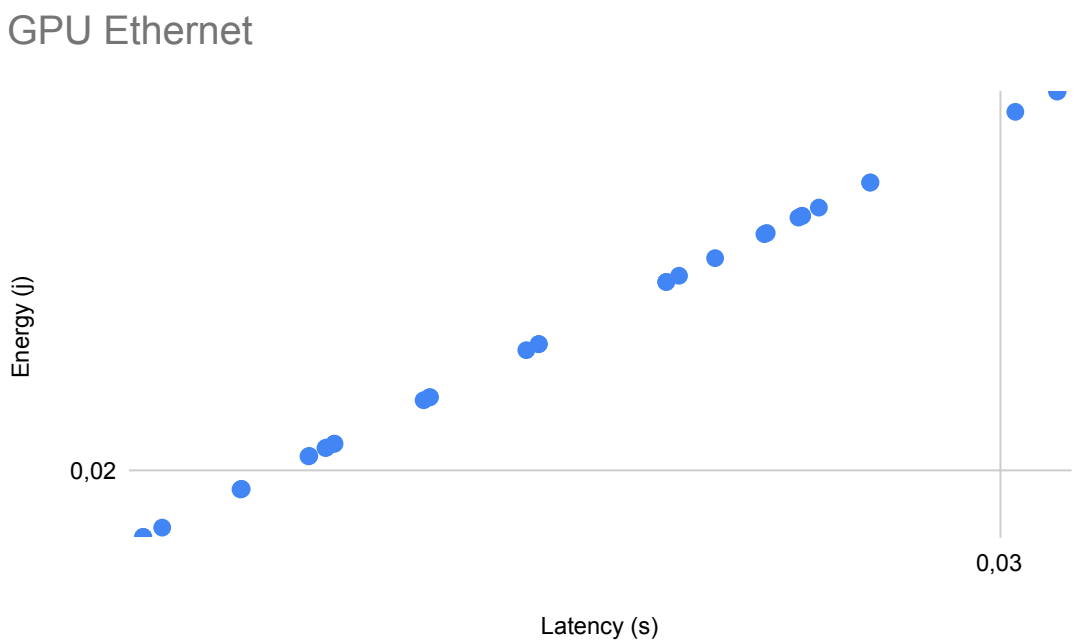


Figure A.10: Correlation between energy and latency for the embedded NPU. Both axis are logarithmic.

A.3 Offloading contributions

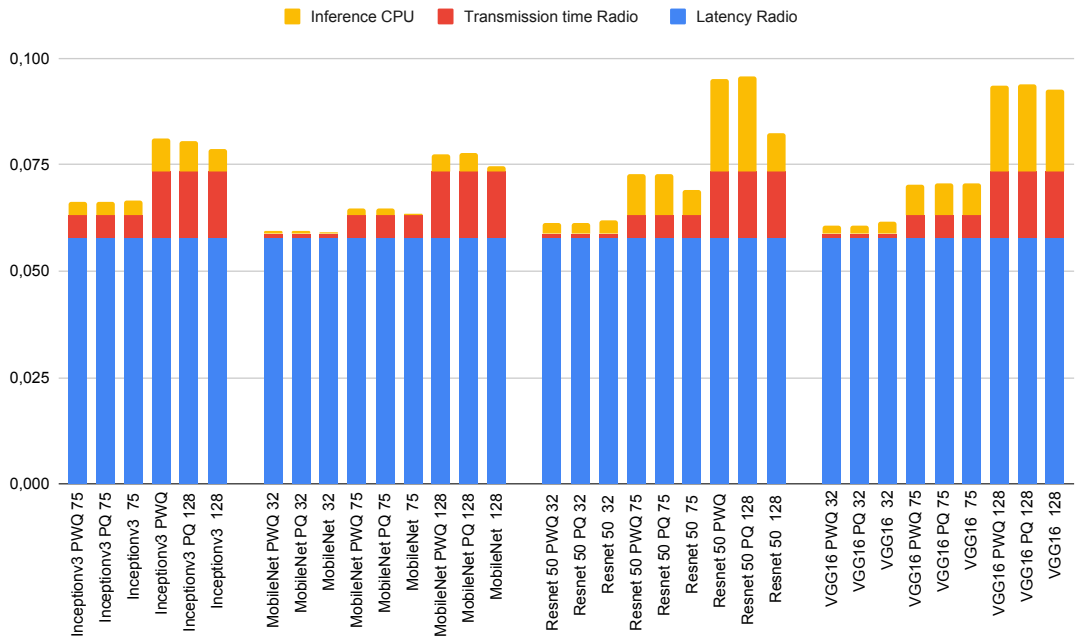


Figure A.11: Illustration of latency contributions of network communication and execution time of model, using radio to offload to CPU

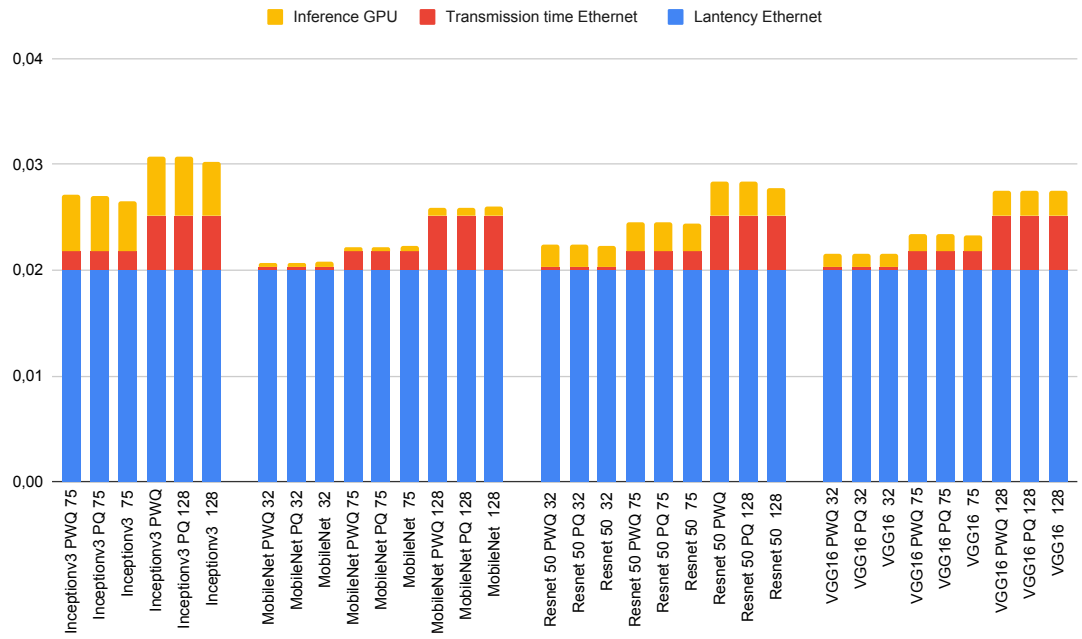


Figure A.12: Illustration of latency contributions of network communication and execution time of model, using Ethernet to offload to GPU

A.4 5G

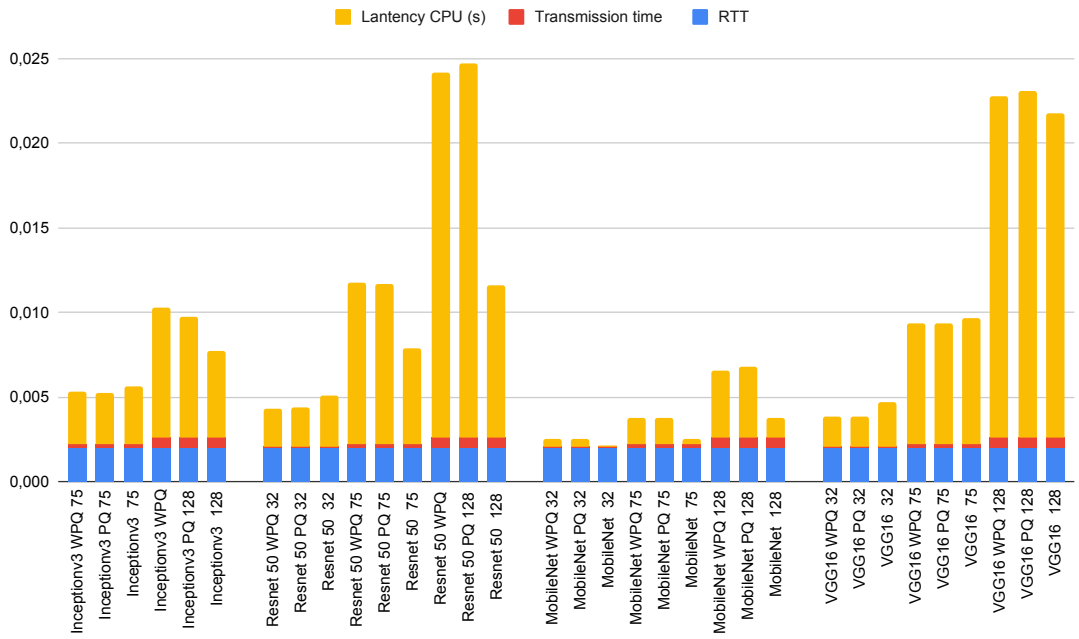


Figure A.13: Illustration of latency contributions of network communication and execution time of model, using 5G to offload to CPU

A.5 Compression Technique Isolation

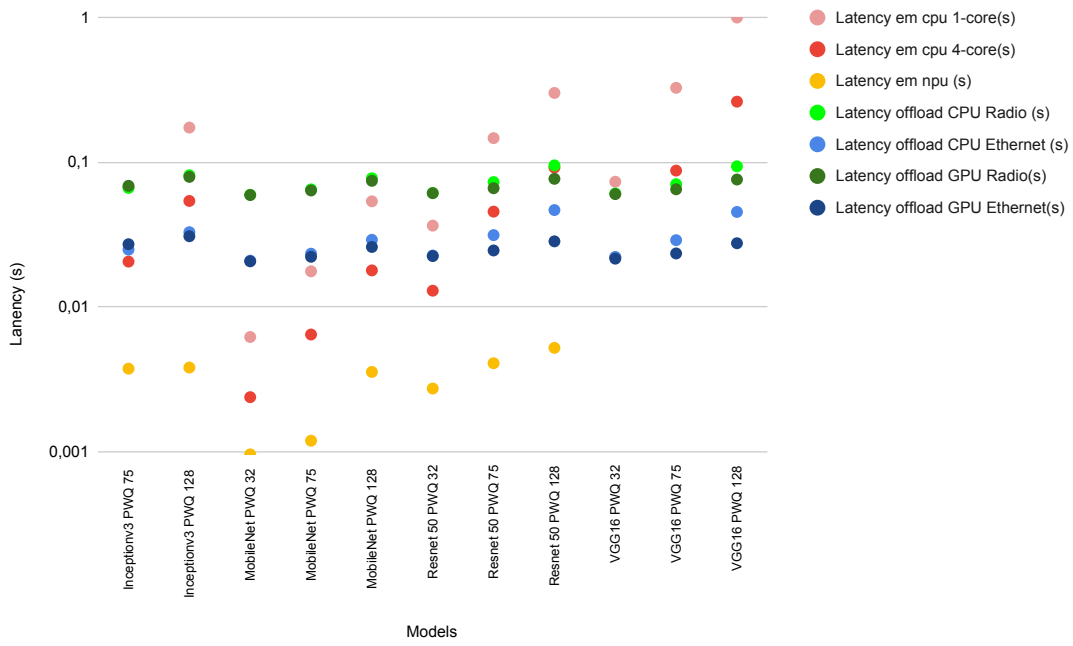


Figure A.14: Latencies of tested models featuring only the variations compressed with Pruning, Weight clustering and quantization.

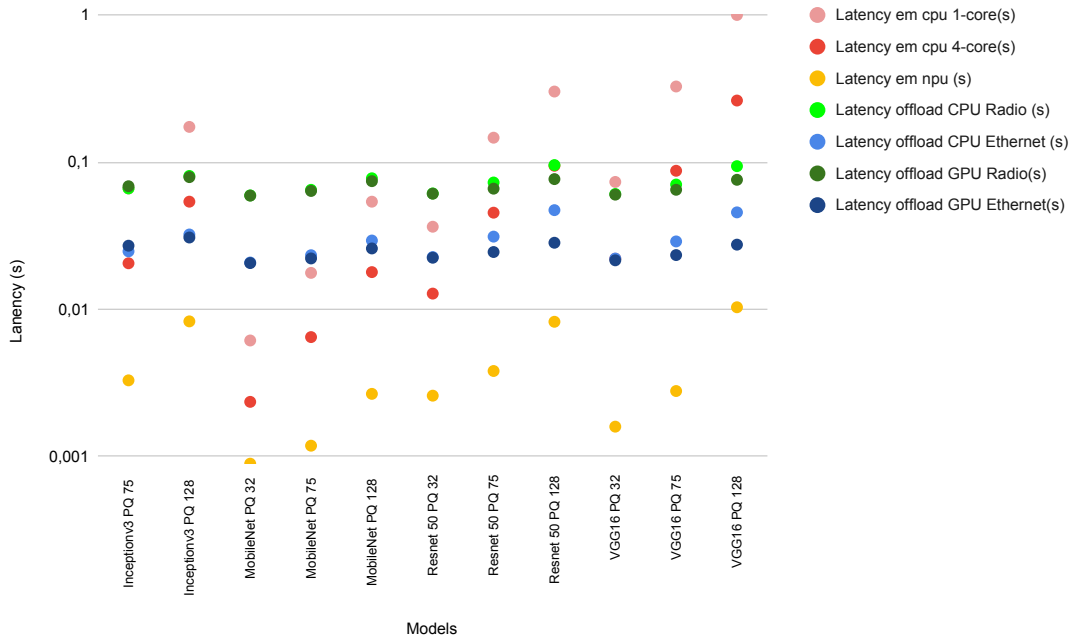


Figure A.15: Latencies of tested models featuring only the variations compressed with Pruning and quantization.

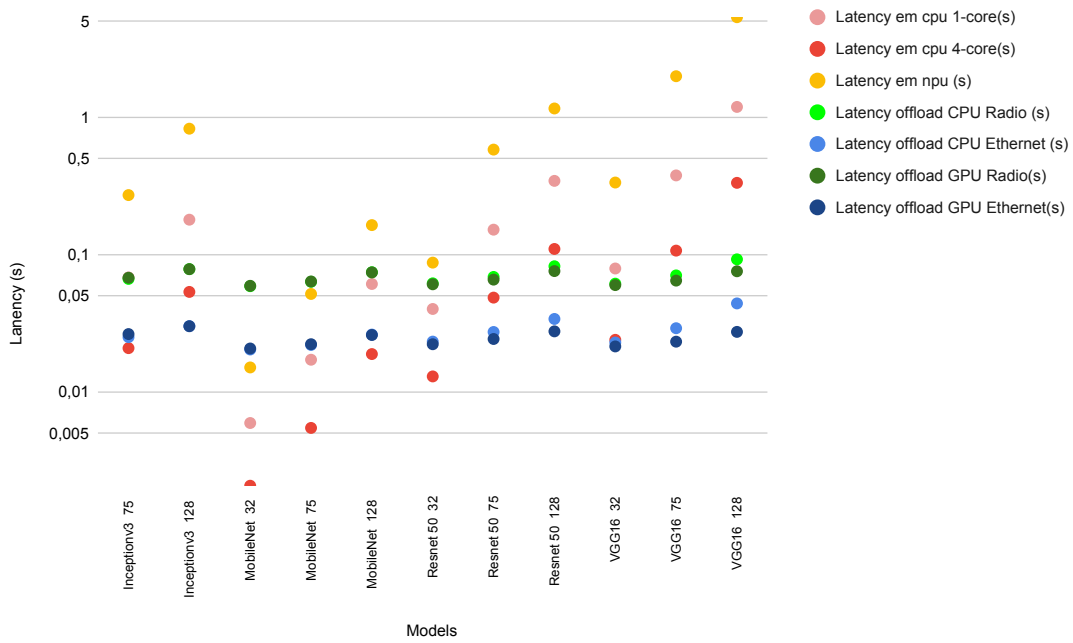


Figure A.16: Latencies of tested models featuring only the uncompressed variations.

A.6 Perf Results

A.6.1 Instructions

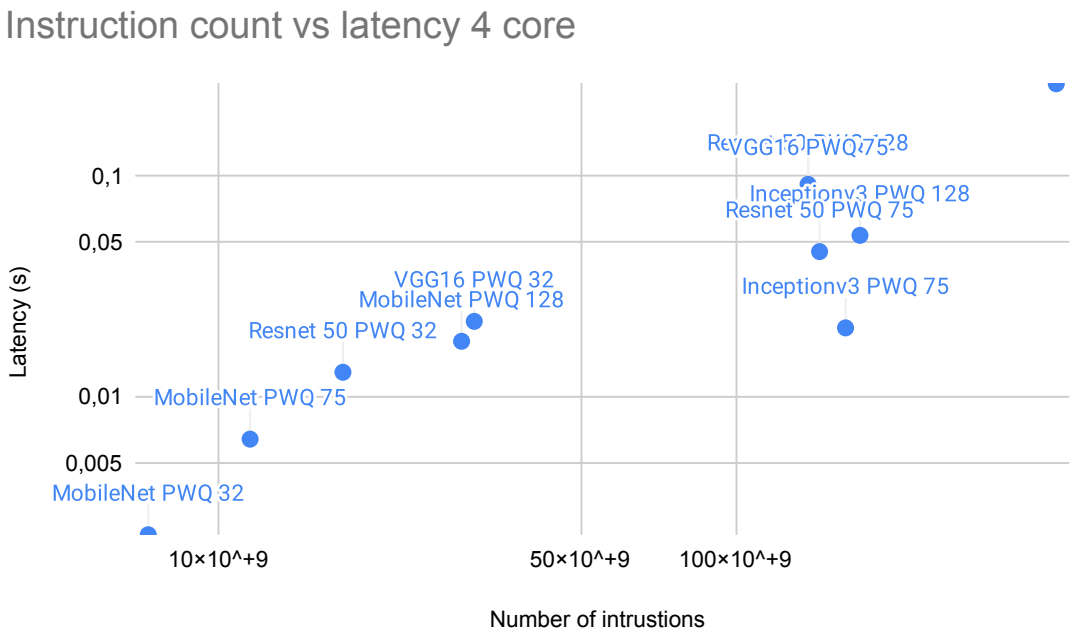


Figure A.17: Number of instructions plotted against the latency on 4 cores for receptive model. Both axis are logarithmic

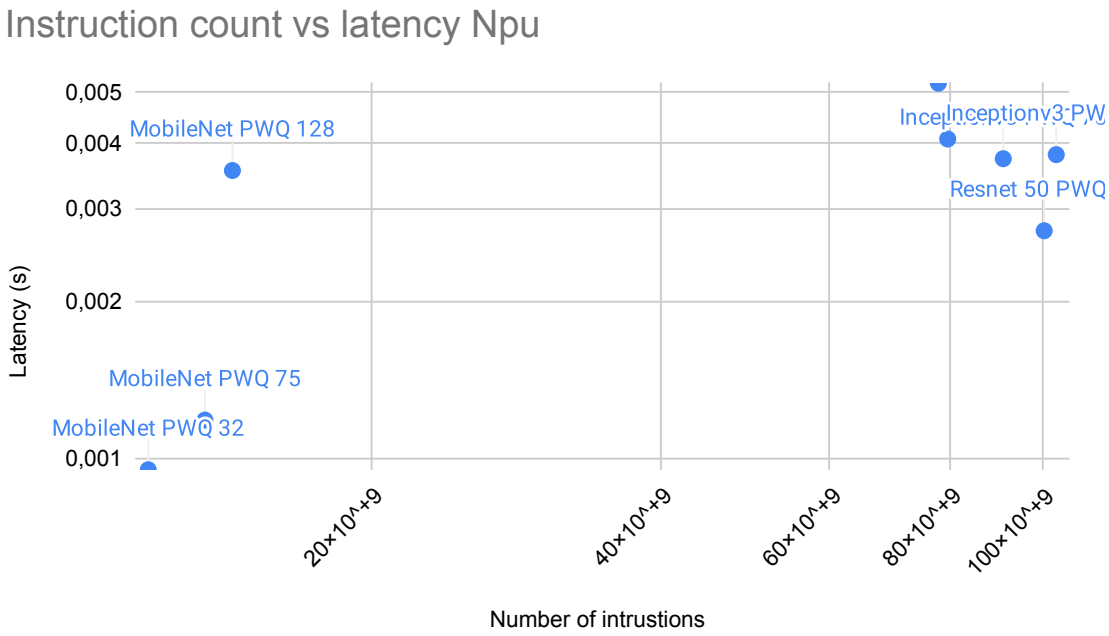


Figure A.18: Number of instructions plotted against the latency on NPU for receptive model. Both axis are logarithmic

A.6.2 L1d cache loads

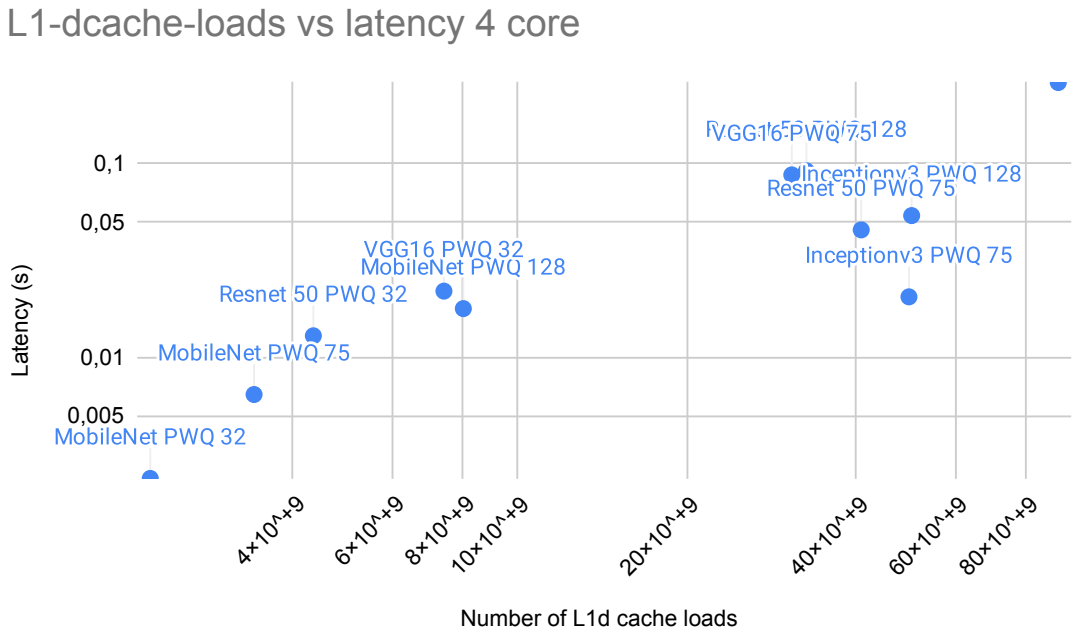


Figure A.19: Number of l1d cache loads plotted against the latency on 4 cores for receptive model. Both axis are logarithmic

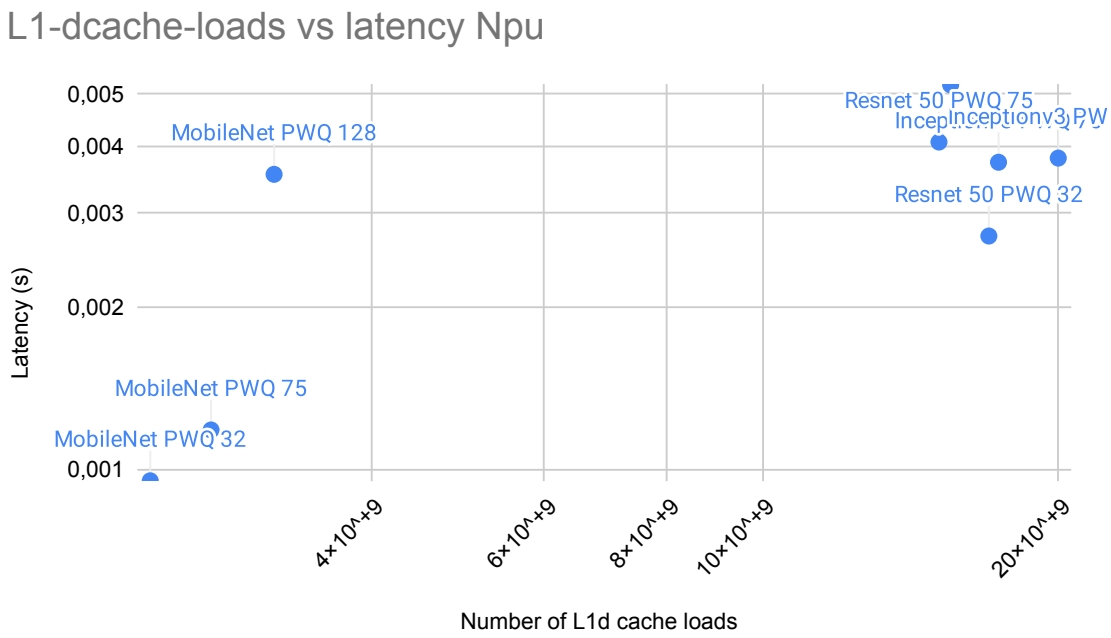


Figure A.20: Number of l1d cache loads plotted against the latency on NPU for receptive model. Both axis are logarithmic

A.6.3 L1 dcache misses

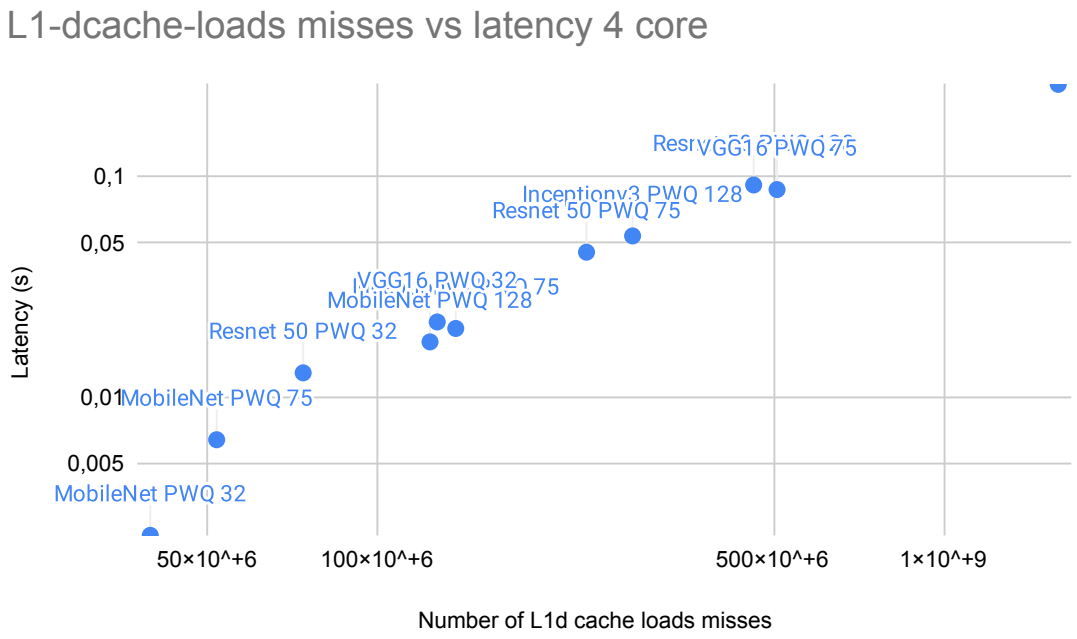


Figure A.21: Number of l1d cache loads misses plotted against the latency on 4 cores for receptive model. Both axis are logarithmic

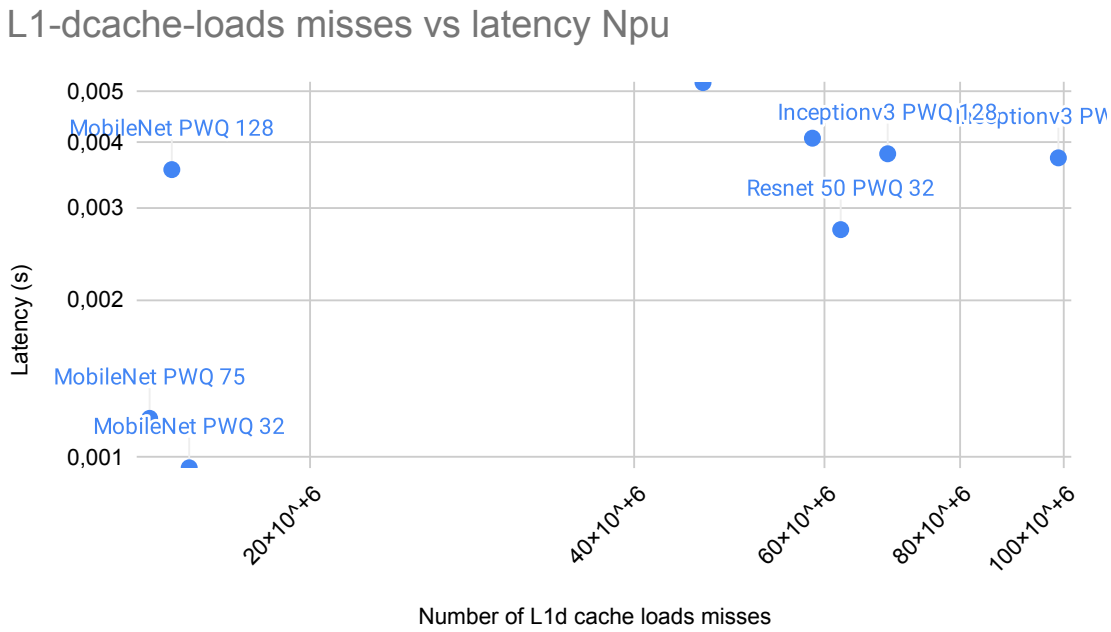


Figure A.22: Number of l1d cache loads misses plotted against the latency on NPU for receptive model. Both axis are logarithmic

A.6.4 Page Faults

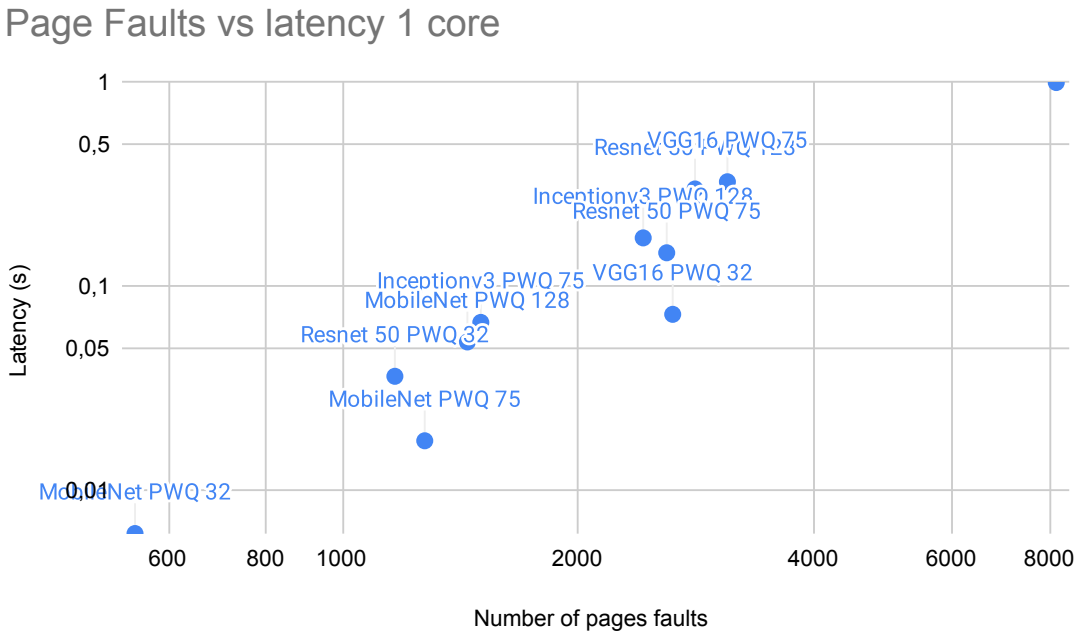


Figure A.23: Number of page faults misses plotted against the latency on 1 core for receptive model. Both axis are logarithmic

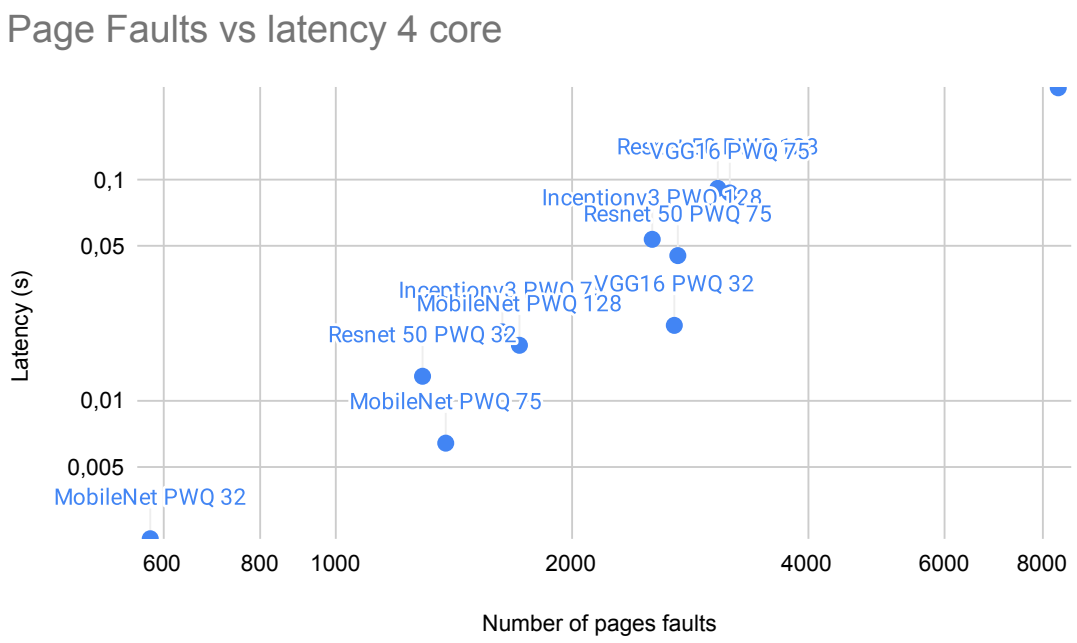


Figure A.24: Number of page faults misses plotted against the latency on 4 cores for receptive model. Both axis are logarithmic

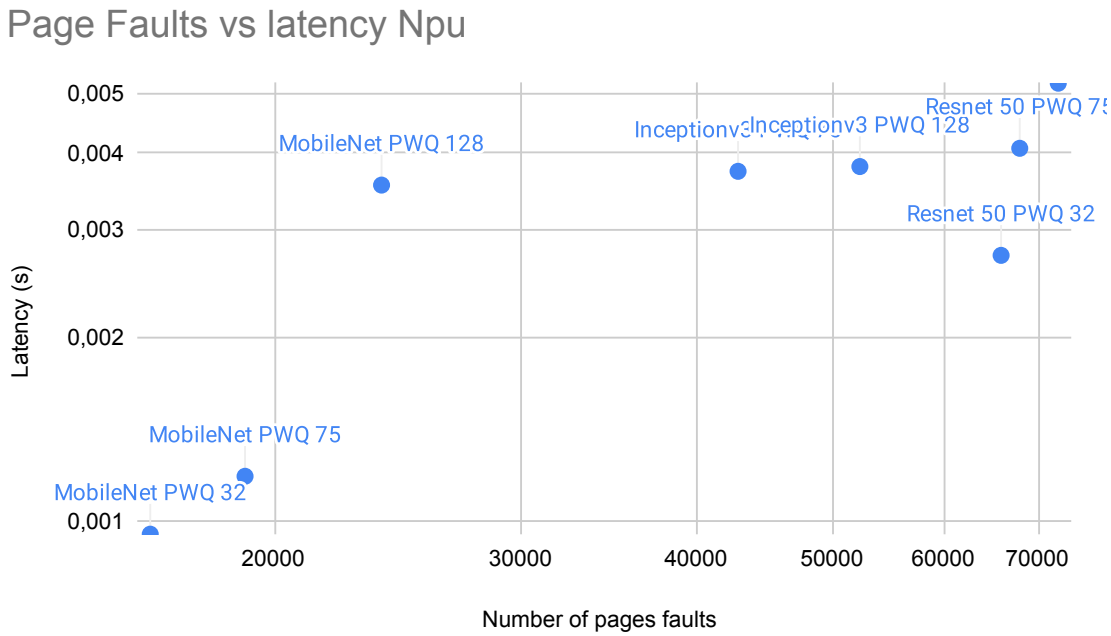


Figure A.25: Number of page faults misses plotted against the latency on NPU for receptive model. Both axis are logarithmic

A.6.5 Instruction vs parameters

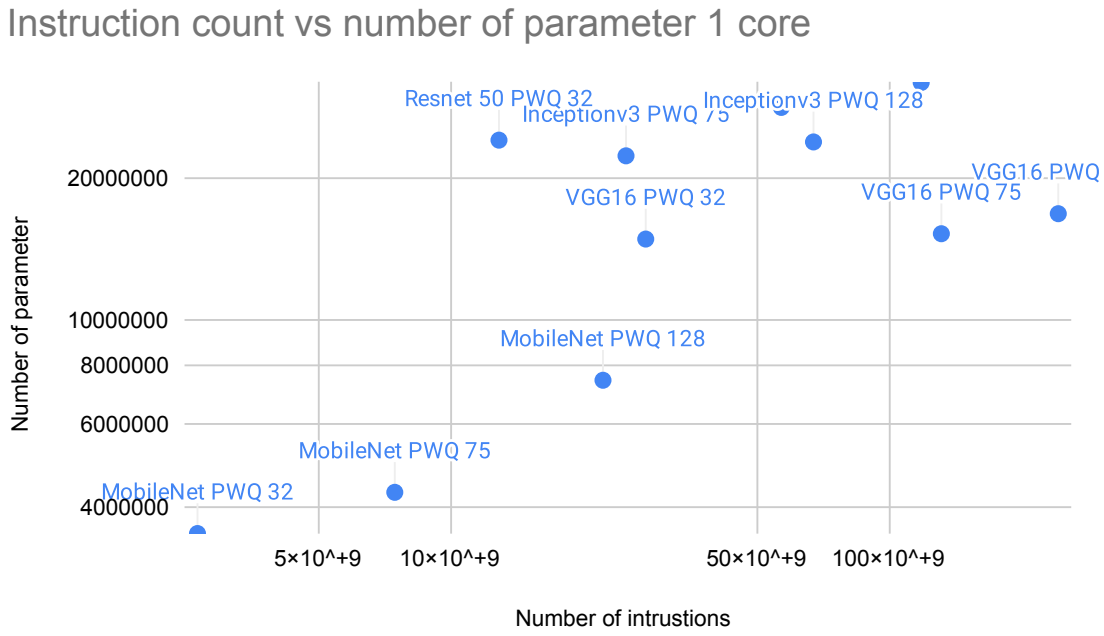


Figure A.26: Number of parameters plotted against the number of instructions on 1 core for receptive model. Both axis are logarithmic

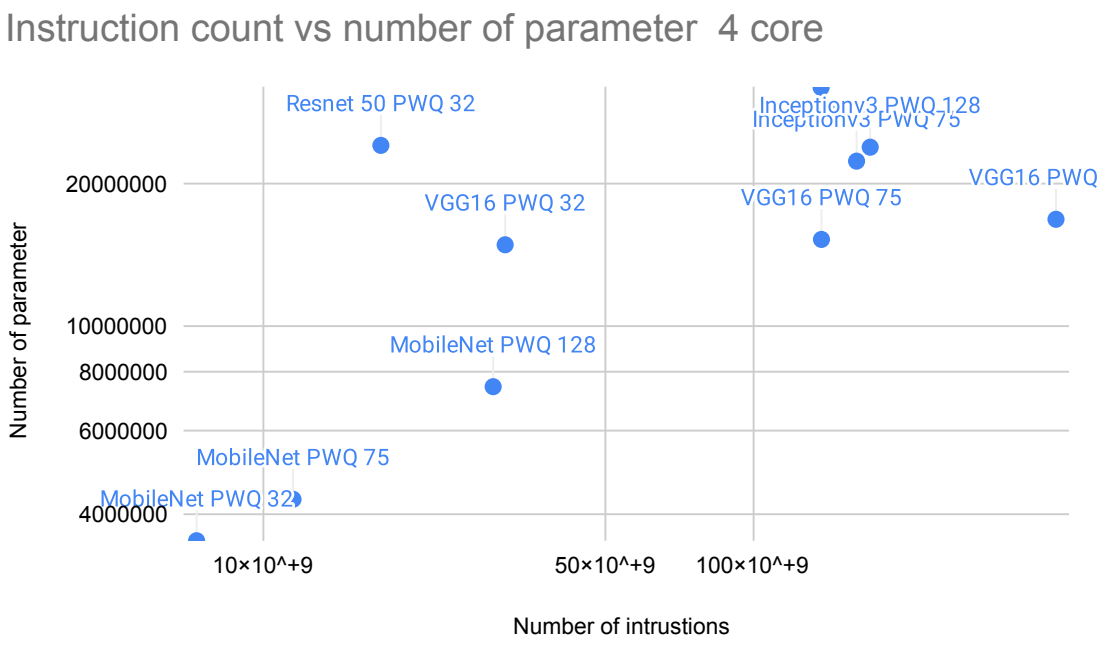


Figure A.27: Number of parameters plotted against the number of instructions on 4 cores for receptive model. Both axis are logarithmic

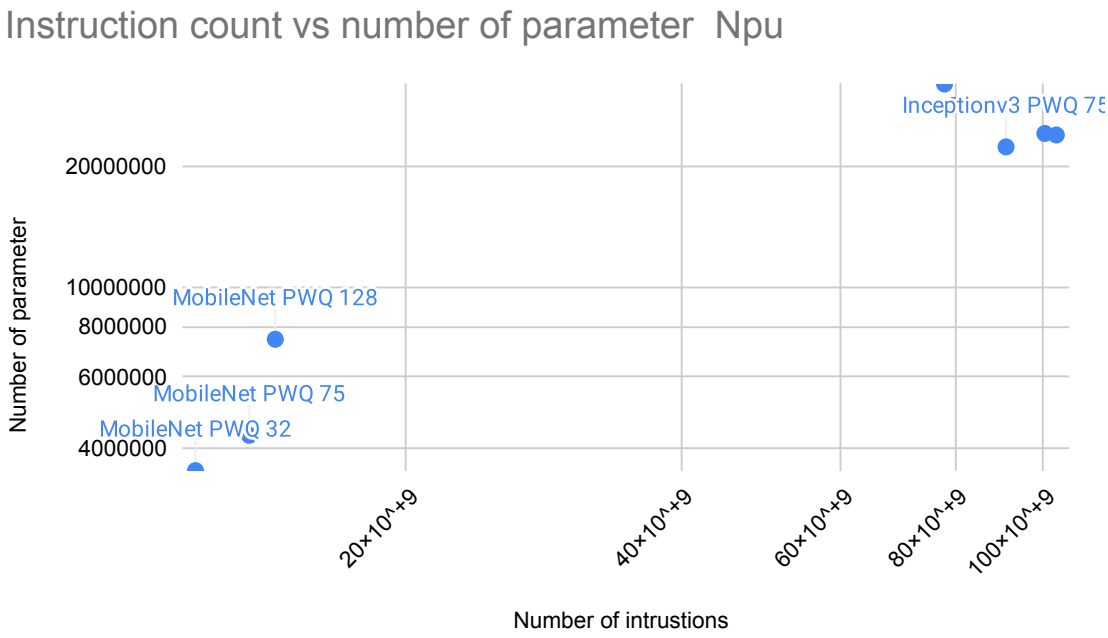


Figure A.28: Number of parameters plotted against the number of instructions on NPU for receptive model. Both axis are logarithmic

A.6.6 L1-dcache-loads vs parameters

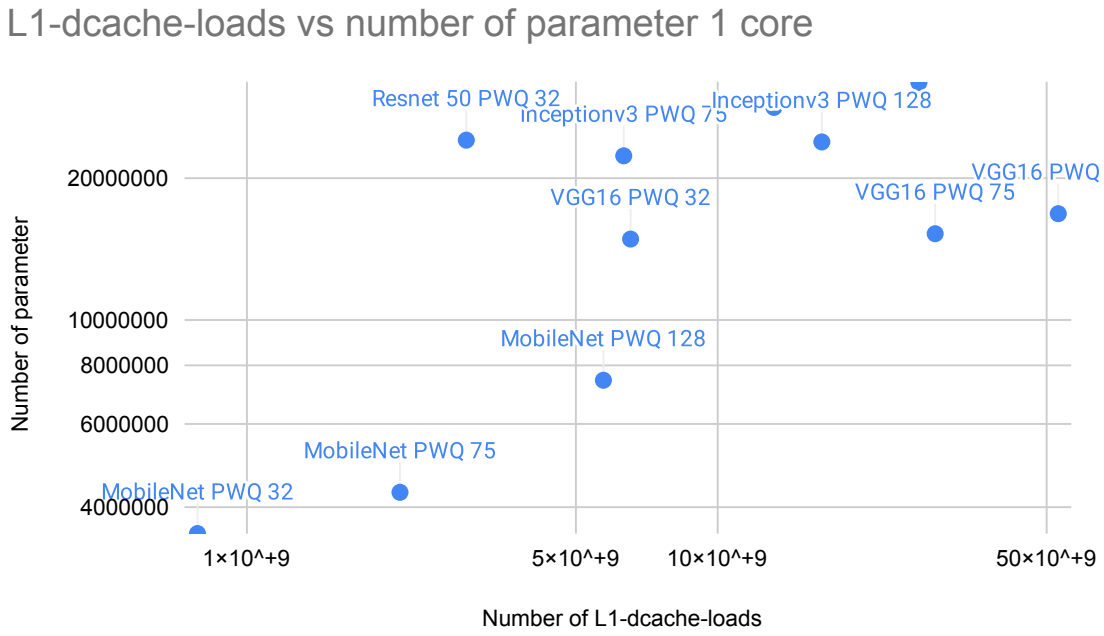


Figure A.29: Number of parameters plotted against the number of L1-dcache-loads on 1 core for receptive model. Both axis are logarithmic

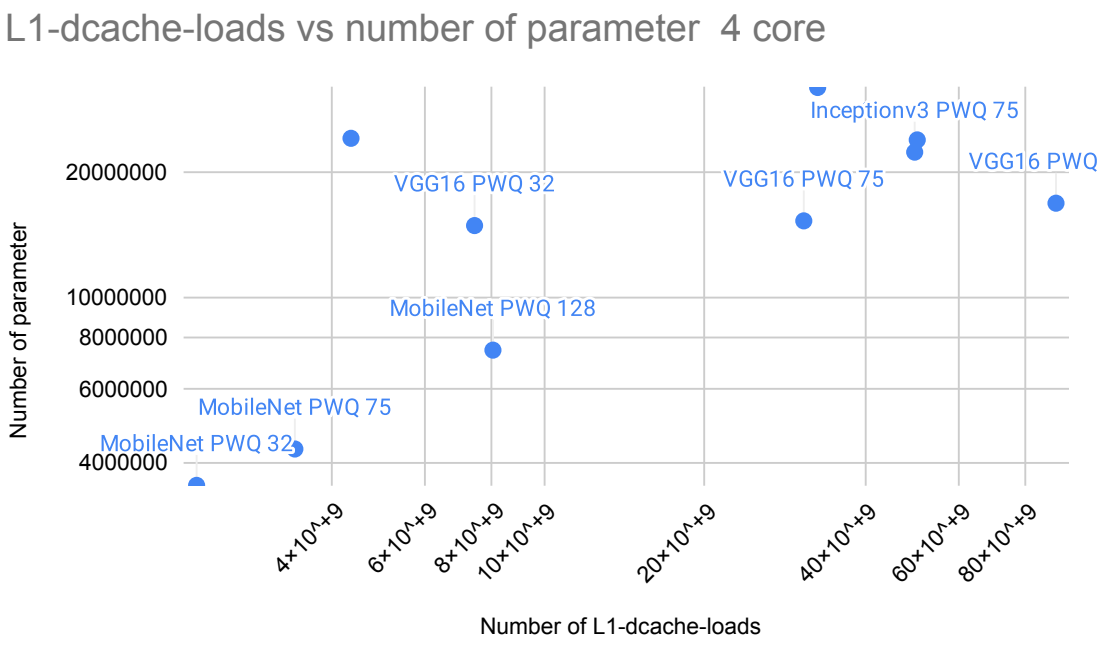


Figure A.30: Number of parameters plotted against the number of L1-dcache-loads on 4 cores for receptive model. Both axis are logarithmic

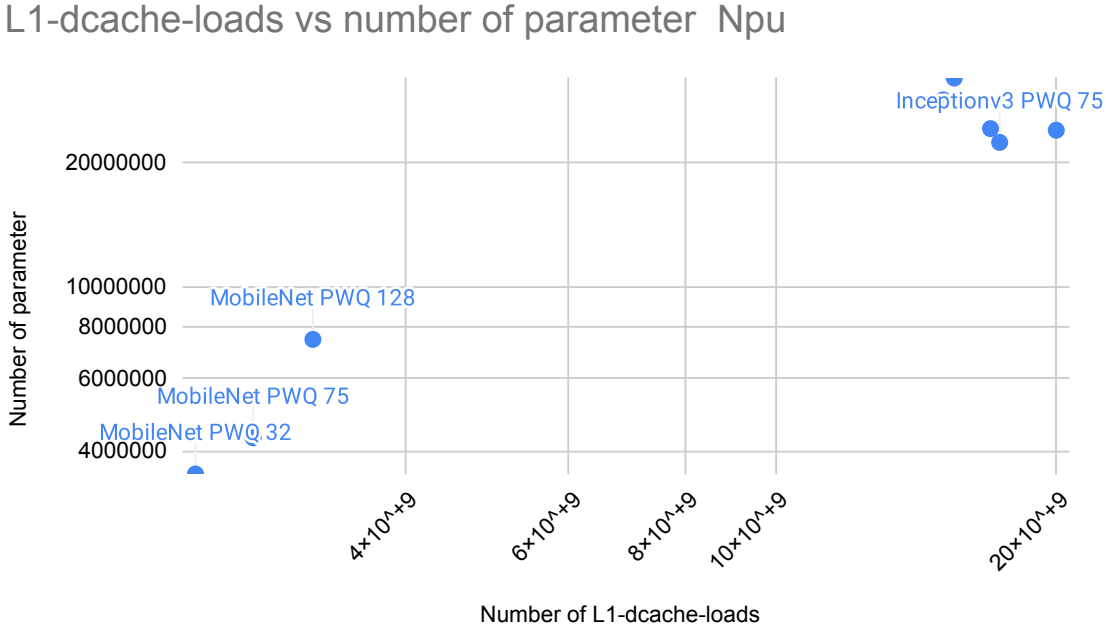


Figure A.31: Number of parameters plotted against the number of L1-dcache-loads on NPU for receptive model. Both axis are logarithmic

B

Yocto Manifest

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote name="CAF"
    fetch="https://source.codeaurora.org/external/imx"/>
  <remote name="OSSystems" fetch="https://github.com/OSSystems"/>
  <remote name="QT5" fetch="https://github.com/meta-qt5"/>
  <remote name="Timesys" fetch="https://github.com/TimesysGit"/>
  <remote name="clang" fetch="https://github.com/kraj"/>
  <remote name="community" fetch="https://github.com/Freescale"/>
  <remote name="imx-support"
    fetch="https://source.codeaurora.org/external/imxsupport"/>
  <remote name="oe" fetch="https://github.com/openembedded"/>
  <remote name="python2" fetch="https://git.openembedded.org"/>
  <remote name="rust" fetch="https://github.com/meta-rust"/>
  <remote name="yocto" fetch="https://git.yoctoproject.org/git"/>

  <default sync-j="2"/>

  <project name="fsl-community-bsp-base" path="sources/base"
    remote="community" revision="5a551f453260bd19895e4d847877874eaa51fde3">
    <linkfile src="README" dest="README"/>
    <linkfile src="setup-environment" dest="setup-environment"/>
  </project>
  <project name="meta-browser" path="sources/meta-browser"
    remote="OSSystems" revision="cb3278e31340c7f081e8deb0683df2145da515c9"/>
  <project name="meta-clang" path="sources/meta-clang"
    remote="clang" revision="d797409435d3b0e9f2859992439989ff1d81e66d"/>
  <project name="meta-freescale" path="sources/meta-freescale"
    remote="community" revision="80dbe4bd63bd537fc9cfda2e009f8543464b4698"/>
  <project name="meta-freescale-3rdparty" path="sources/meta-freescale-3rdparty"
    remote="community" revision="7f23af99cb97a12134a46b5b9d497f05b758bf0c"/>
  <project name="meta-freescale-distro" path="sources/meta-freescale-distro"
    remote="community" revision="916df6d24c0a33a3b1533bde70b6a2724ec77af4"/>
  <project name="meta-imx" path="sources/meta-imx" remote="CAF"
    revision="refs/tags/rel_imx_5.10.35_2.0.0" upstream="hardknott-5.10.35-2.0.0">
    <linkfile src="tools/imx-setup-release.sh" dest="imx-setup-release.sh"/>
    <linkfile src="README" dest="README-IMXBSP"/>
```

```
  </project>
  <project name="meta-nxp-demo-experience" path="sources/meta-nxp-demo-experience"
    remote="imx-support" revision="46107357abd2d2da9ffd702c87fce3984a422435"
    upstream="imx_5.10.y" dest-branch="imx_5.10.y"/>
  <project name="meta-openembedded" path="sources/meta-openembedded"
    remote="oe" revision="c3a36263f91e42302ad7c347e051cf1cd83e39f6"/>
  <project name="meta-python2" path="sources/meta-python2"
    remote="python2" revision="810d6d842f103eb59f18b06426106462b15de7e2"/>
  <project name="meta-qt5" path="sources/meta-qt5"
    remote="QT5" revision="a00af3eae082b772469d9dd21b2371dd4d237684"/>
  <project name="meta-timesys" path="sources/meta-timesys"
    remote="Timesys" revision="00f81fbdf7fba2a09ff83d14fc3b040e9ae63b42"/>
  <project name="poky" path="sources/poky" remote="yocto"
    revision="58cbdaecf75b0248f96780b6882e8d4f232d038a"/>
</manifest>
```