FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Tuomas Tuokkola

LATENCY AND ACCURACY OPTIMIZED MOBILE FACE DETECTION

Master's Thesis
Degree Programme in Computer Science and Engineering
December 2020

Tuokkola T. (2020) Latency and Accuracy Optimized Mobile Face Detection. University of Oulu, Degree Programme in Computer Science and Engineering, 69 p.

ABSTRACT

Face detection is a preprocessing step in many computer vision applications. Important factors are accuracy, inference duration, and energy efficiency of the detection framework. Computationally light detectors that execute in real-time are a requirement for many application areas, such as face tracking and recognition. Typical operating platforms in everyday use are smartphones and embedded devices, which have limited computation capacity.

The capability of face detectors is comparable to the ability of a human in easy detection tasks. When the conditions change, the challenges become different. Current challenges in face detection include atypically posed and tiny faces. Partially occluded faces and dim or bright environments pose challenges for detection systems. State-of-the-art performance in face detection research employs deep learning methods called neural networks, which loosely imitate the mammalian brain system. The most relevant technologies are convolutional neural networks, which are designed for local feature description.

In this thesis, the main computational optimization approach is neural network quantization. The network models were delegated to digital signal processors and graphics processing units. Quantization was shown to reduce the latency of computation substantially. The most energy-efficient inference was achieved through digital signal processor delegation. Multithreading was used for inference acceleration. It reduced the amount of energy consumption per algorithm run.

Keywords: energy-efficiency, face detection, smartphones, real-time, quantization, deep learning, convolutional neural networks, RetinaFace

Tuokkola T. (2020) Latenssi- ja tarkkuusoptimoitu kasvontunnistus mobiililaitteilla. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 69 s.

TIIVISTELMÄ

Kasvojen ilmaisu on esikäsittelyvaihe monelle konenäön sovellukselle. Tärkeitä kasvoilmaisimen ominaisuuksia ovat tarkkuus, energiatehokkuus ja suoritusnopeus. Monet sovellukset vaativat laskennallisesti kevyitä ilmaisimia, jotka toimivat reaaliajassa. Esimerkkejä sovelluksista ovat kasvojen seuranta- ja tunnistusjärjestelmät. Yleisiä käyttöalustoja ovat älypuhelimet ja sulautetut järjestelmät, joiden laskentakapasiteetti on rajallinen.

Kasvonilmaisimien tarkkuus vastaa ihmisen kykyä helpoissa ilmaisuissa. Nykyiset ongelmat kasvojen ilmaisussa liittyvät epätyypillisiin asentoihin ja erityisen pieniin kasvokokoihin. Myös kasvojen osittainen peittyminen, ja pimeät ja kirkkaat ympäristöt, vaikeuttavat ilmaisua. Neuroverkkoja käytetään tekoälyjärjestelmissä, joiden lähtökohtana on ollut mallintaa nisäkkäiden aivojen toimintaa. Konvoluutiopohjaiset neuroverkot ovat erikoistuneet paikallisten piirteiden analysointiin.

Tässä opinnäytetyössä käytetty laskennallisen optimoinnin menetelmä on neuroverkkojen kvantisointi. Neuroverkkojen ajo delegoitiin digitaalisille signaalinkäsittely- ja grafiikkasuorittimille. Kvantisoinnin osoitettiin vähentävän laskenta-aikaa huomattavasti ja suurin energiatehokkuus saavutettiin digitaalisen signaaliprosessorin avulla. Suoritusnopeutta lisättiin monisäikeistyksellä, jonka havaittiin vähentävän energiankulutusta.

Avainsanat: energiatehokkuus, kasvoilmaisu, älylaitteet, reaaliaikainen, kvantisointi, syväoppiminen, konvoluutioneuroverkot, RetinaFace

TABLE OF CONTENTS

FOREWORD

I want to thank everyone at Visidon for their good team spirit and emphasis on relaxed but efficient working culture. I want to give my greatest gratitude to Henri Nykänen, Olli Silven, Jari Hannuksela, Jukka Holappa and Tuomas Holmberg.

Henri Nykänen was my technical supervisor at the company and without Henri's original work and help I would have been, to large extent, lost in time. Olli Silven was my thesis supervisor at University and helped me with the scoping of the thesis and in theoretical writing. Jari Hannuksela was the second examiner of the thesis and helped with the power dissipation test setup. Jukka Holappa and Tuomas Holmberg were my roommates in the old office (from January of 2020 to May of 2020) and helped me to grow professionally. Sometimes with some implementation tips and in-depth questions about work, but sometimes also of life. Special thanks to Tuomas Holmberg for helping me with the challenges with the LaTeX-template of this thesis.

I want to show my gratitude to everyone who has contributed to this thesis through writing clear and unambiguous blog posts and books on related subjects and to everyone who has been mentioned in these blog posts and books as contributors of any sort.

Oulu, December 8th, 2020

Tuomas Tuokkola

LIST OF ABBREVIATIONS AND SYMBOLS

ReLU	rectified linear unit
CNN	convolutional neural network
PTQ	post-training quantization
DSP	digital signal processor
TPU	tensor processing unit
MAC	multiply accumulate operation
FP	floating point
FP-16	16-bit floating point
FP-32	32-bit floating point
FX	fixed point
FX-8	8-bit fixed point
MM	matrix multiplication
FFT	fast fourier transform
SIMT	single instruction, multiple threads
MIMT	multiple instructions, multiple threads
CPU	central processing unit
GPU	graphics processing unit
IoU	intersect over union
NMS	non-maximum suppression
QVGA	quarter video graphics array
VGA	video graphics array
FHD	full high definition
4K DCI	4K digital cinema initiatives
PR	precision-recall
FPR	false positive rate
FNR	false negative rate
DC	direct current
T1	single thread
TB	balanced threading
TT	maximum throughput threading
σ^2	batch variance
μ	batch mean
γ	standard deviation parameter
β	mean parameter
ϵ	epsilon
z or X_z	zero point
s or X_s	scaling factor
q or X_q	quantized value
$\boldsymbol{T}$	tensor
$\boldsymbol{M}$	matrix
$\boldsymbol{v}$	vector
s	scalar

1. INTRODUCTION

Computer vision is a field of artificial intelligence in which systems are designed to interpret the visual world. Due to the importance of vision, the human body has developed a vast system for processing and interpreting visual signals that the research tries to imitate. [1]

With early face detection algorithms, slight differences in the orientation of a head could easily affect the prediction outcome. Another goal was low latency. For some real-world applications the processing should be done in real-time. [2]

Face detection deals with a variety of challenges. Datasets are used for categorization of known problem cases. Figure 1 represents core challenges in WIDER Face dataset. [3]

Figure 1. Datasets contain various challenges for face detection algorithms.

Decades ago computers were lacking the computational resources needed for processing useful artificial neural networks. Although some systems were introduced, not many were effectively put in use. Development of hardware efficiency and power output in coming years changed the situation.

Multi-scale detection is one of many active object detection research areas. Finding small faces in large images has been a complex issue in face detection research. An example of a challenging multi-scale detection situation is presented in Figure 2. [4]

Figure 2. World's largest selfie by Lumia 730 is a challenging face detection scenario.

The use of smartphones introduced new challenges for face detection algorithm design. Energy efficiency became important, while neural computing-based solutions require traditionally vast amounts of computation. The majority of neural computation is multiply-accumulate operations. The needed operations are simple but the problem is in the quantity. Optimization techniques designed for dealing with this computation bottleneck have branched in many directions. [2]

Common optimization solutions for neural computing can be roughly categorized followingly:

- Speeding-up the detection pipeline

  Speeding-up the detection pipeline is achieved by implementing multiply-accumulate computation on novel specialized hardware. The most common delegation hardware are graphics processing units (GPUs) and digital signal processors (DSPs). Specialized hardware is more efficient in terms of energy usage.

- Numerical precision reduction

  Optimization methods introduce a level of error to classification accuracy. The precision reduction is called quantization. The predictive power of a neural network is based on the idea of receptive fields, which capture both large- and small-scale structures. Due to a huge amount of variables, most networks are tolerant of error.

- Algorithm innovations

  Lightweight deep learning methods are used for energy-efficient face detection. Mobile deep learning algorithms are the optimal choice for a trade-off in terms of prediction accuracy and detection latency.

1.1. Scope of the Thesis

In Chapter 2 traditional face detection systems are considered. Main mechanisms and elements of neural computing are covered and the most relevant neural network optimization techniques are elaborated. In Chapter 3 the fundamentals of computer arithmetics and computing models are described. The architectures are considered in neural computing context. In Chapter 4 a face detection implementation is presented. In Chapter 5 a set of measurements are conducted and metrics are introduced for evaluation purposes. Chapter 6 discusses topics that are relevant for the future of this work.

2. FACE DETECTION

Face detection is a relevant phase in various modern problems involving person verification, identification, and expression analysis. The first stage of a face detection system is feature extraction, where representations of features are acquired. Extracted feature representations are used for detection classification.

2.1. Hand-Crafted Features Based Methods

Face detection is traditionally based on using hand-crafted features. The most common feature forms are edges and corners. Features are used for finding proportionally distinct intensity changes within image presentation [2]. For instance, Harris corner detector is an algorithm where edge and corner feature representation was combined into a single detector [5]. Early detector pipelines usually consisted of noise reduction, feature calculation, thresholding, and normalization algorithms [6]. Figure 3 visualizes the features used by a Canny edge detector. Image taken from [3].

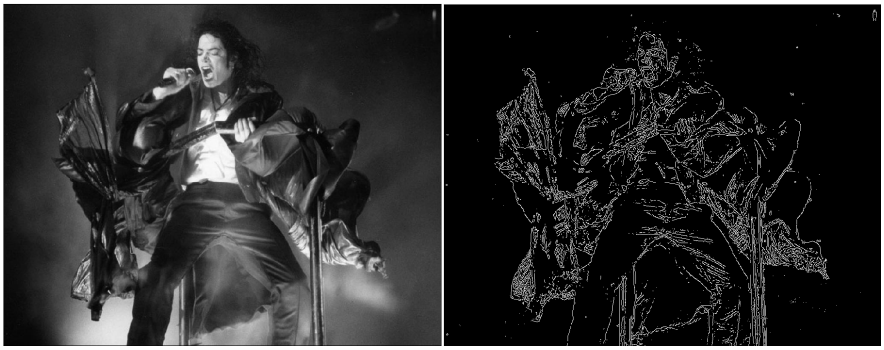

Figure 3. Example of an edge detection algorithm.

Feature variance is relevant challenge in object detection systems. Scale-invariant features (SIFT) is a traditional algorithm for scale and variance tolerant feature extraction. Scale, translation, and rotation invariant features are expensive to calculate due to image information richness. [7]

For latency minimization speeded-up robust features (SURF) algorithm was developed for efficient GPU execution. Both SURF and SIFT are based on local information, and on features calculated from gradients in an image [8]. Another traditionally popular algorithm is histogram of oriented gradients (HOG), the flow of which is displayed in Figure 4 [9].

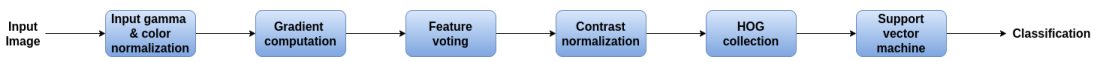

Figure 4. HOG detector workflow.

The basic face detection methods can be categorized into template-matching and part-based model schemes. Template-matching methods introduced a face template, which can contain dimensions, contour, and facial elements. Elements can contain

knowledge about relationships present in a face or a set of facial features. This approach is simple to implement, but performs poorly compared to other face detection algorithms. [10]

The idea in part based models is that each object consists of one global base filter and several part models, for example, a head and eyes. Usually the first model, the body, is defined by a coarse template. The model defines the placement space for its child part models, which are usually inspected at a higher resolution. Both body and child filters are scored based on some evaluation rules. These rules have been extended to include multiple methods during the years and a large portion of algorithmic success was due to these advancements. [11]

Image channel features have played a huge part in face detection over the years. Image channel methods are used for non-linear and linear face detection, where the most common features are extracted from color channels [12]. There are also many non-linear channels, and the most popular use of non-linear channels were adapted with a Canny edge detector [6] by displaying edges and gradient magnitude (edge strength) [2].

2.1.1. Viola-Jones

Among the first robust and real-time capable face detection algorithm is the Viola-Jones (VJ) face detector. One of the main contributions was the integral image representation method for accelerated feature evaluation. It is a method for narrowing the total number of needed haar-like features without losing quality from the prediction.

VJ detector used several weak learners for forming a strong classifier for a classification task. A weak learner is a learner that does not hold enough information for a detection alone.

In Figure 5 haar features are visualized. The features are obtained by subtracting the sum of pixels under the white rectangle from pixels under the black rectangle. This way the detector gains information from relevant relationships for a face detection. [13]

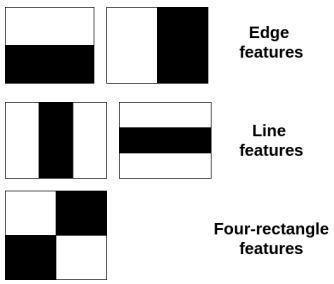

Figure 5. Example of a haar-feature set.

Vast number of predictions in a detection are negative. Detection cascade was developed for dealing with this challenge. The cascading technique increases the performance of a detector significantly by focusing attention on promising image regions. In the VJ detector, a variant of adaptive boosting (Adaboost) is used for

selecting a core set of features and boosting the classification performance of a system. [14]

2.1.2. LBP Based Face Detection

Local binary pattern (LBP) methods are originally general texture descriptor methods. The methods were extended to face detection domain as a rotation invariant and lightweight techniques. LBP considers compositions of micro-patterns as global descriptors and the method is commonly applied on a grey-scale image. LBP operator is calculated using a pixel and its neighbourhoods, originally an eight-pixel neighbourhood method was used. In this method a binary collection of pixels is obtained by starting binary from top-left corner and advancing in clockwise direction. [15]

LBP for face detection relies on extended usage of neighbourhoods, some variations are (8,1) and (16,2) neighbourhood sizes. Later the operator was extended to uniform patterns, which is an pattern, where transition between bits is at most two. [15]

The first step in using LBP for face detection is calculating feature histograms. Figure 6 represents LBP histogram generation stages, in which extracted binary values form histograms of features.

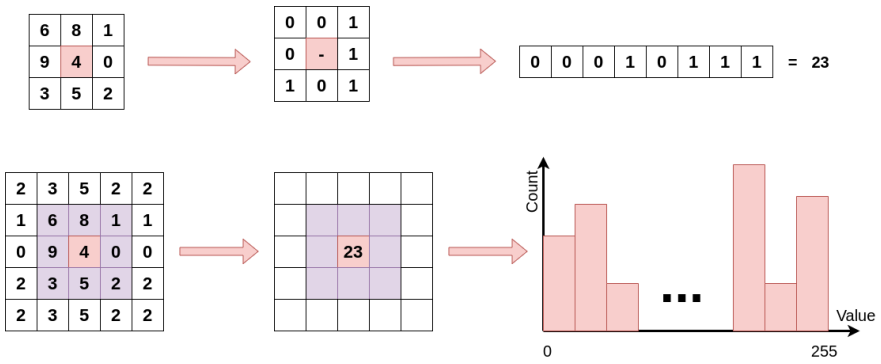

Figure 6. LBP feature generation stages.

Some more recent advances in LBP usage have been adaptations from other traditional detection paradigms: usage of other classification techniques in pre- or post-processing, and efficient usage of a cascade of classifiers. Some applied classification techniques are support vector machine (SVM) and eigenface (PCA) methods. Facial representation is considered as small non-overlapping regions used to form feature histograms for the prediction. [16]

LBP is a lightweight and fast algorithm in terms of feature calculation. However, LBP based features cannot be learned by neural computing.

2.1.3. Feature Learning in Face Detection

Feature learning methods capture efficient feature representations. Two main categories exist: supervised and unsupervised learning. In supervised learning we

have data and labels. The goal is to adjust the parameters so the system finds how the data and labels are correlated based on a loss function. Unsupervised learning works without the labels. [17]

In this thesis we apply supervised learning. Its phases are

- Training

  The learning goal is reached through an iterative training process. Model weights are either tuned or learned in this process.

- Testing

  In testing phase the testing data is used for model evaluation during training. Testing can be used as an evaluation tool for observing the loss function.

- Validation

  The validation data is used for the final validation of the model.

Datasets that consist of data, labels, and noise, are used for feature learning. In a supervised learning environment labels can be used for building a meaningful loss function for the trained network. The network is meant to learn weight parameters from the training data. Testing and validation data sets are often smaller, and are used for determining if the loss converges. The network has not used the testing and validation data for training, so the classification situations are new for the network. [18]

In Figure 7 the design flow of a supervised learning system is depicted.

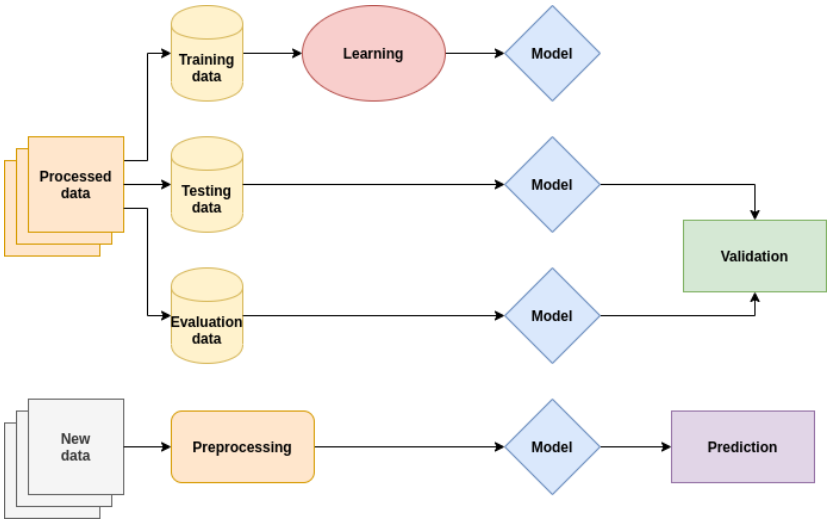

Figure 7. Supervised learning stages.

In training phase the prediction ability of a model is created. The challenges in feature learning are over- and underfitting. Overfitting means that the model follows the details of the data too closely thus being unable to adapt to new situations. Underfitting means that the model has not been able to capture patterns present in the data. [19]

In Figure 8 model fitting challenges are visualized. The system needs to be able to find essential patterns in the data for optimal detection capability. [18]

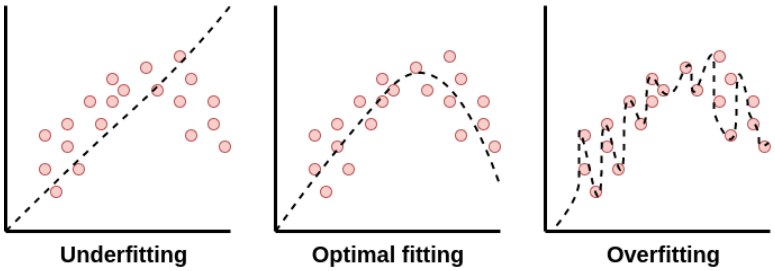

Figure 8. Challenges present in fitting of a Machine Learning model.

Neural networks are usually trained with backpropagation algorithm. Feedforward algorithm is used for predicting. [17]

2.2. Feedforward Algorithm

In feedforward, models the information flow is directed from input layers through the hidden layers to the output layers. Layers consist of neurons that get their inputs from the output signals of the preceding layer, forming a chain of processes. The output layer of the network constitutes the overall response to the activation pattern fed by the input layer [20]. A neuron is visualized in Figure 9, where a node consist of n connections, input values $\boldsymbol{x} = \{x_0, \ldots, x_n\}$, synapse weights $\boldsymbol{w} = \{w_0, \ldots, w_n\}$, a bias b, and an activation function f. Figure is adapted from [17].

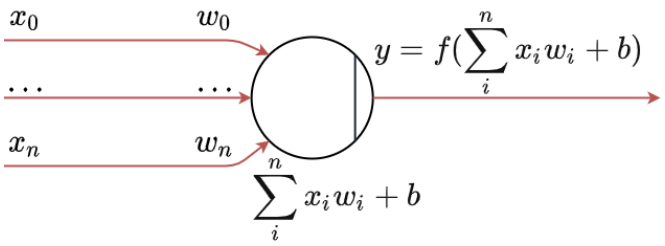

Figure 9. Early prototype of a neuron.

Feedforward algorithm is the backbone of neural computing. In Figure 10 a feedforward system is presented, consisting of input, hidden, output, and activation nodes, which define what connections are active during neural computing.

Activation functions enable networks to learn complex data. Nonlinear activation function rectified linear unit (ReLU) and its variations are often used in the state-of-the-art (SOTA) classification networks. The algorithm of ReLU is given in Equation (1). ReLU introduces non-linearity by mapping negative values as zeros. In other cases input values remain intact. [21]

$$f(x) = max(x, 0) \tag{1}$$

Due to its simplicity ReLU has small computation cost, and negative values do not form connections. [22]

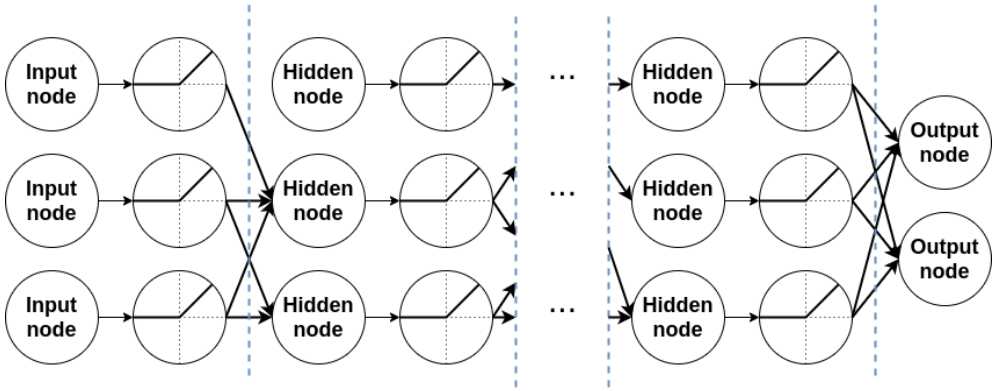

Figure 10. Feedforward algorithm is the backbone of a neural network.

Convolutional neural networks (CNNs) are used for SOTA face detection feature calculation. CNNs employ multistage texture extraction. Feature learning can be accelerated by use of preprocessing techniques, such as batch normalizations. [23]

2.2.1. Preprocessing

Feature scaling enables faster and stabler learning. As we progress deeper in the network, the input parameters for each layer shift in the distribution of inputs, slowing down the optimization. This is known as internal covariate shift. [23]

Parameter whitening has been used for accelerating learning. In whitening the mean and variance of each feature is normalized to be the same. Figure 11 illustrates the principles: The range of features is transformed into the same scale. [23]

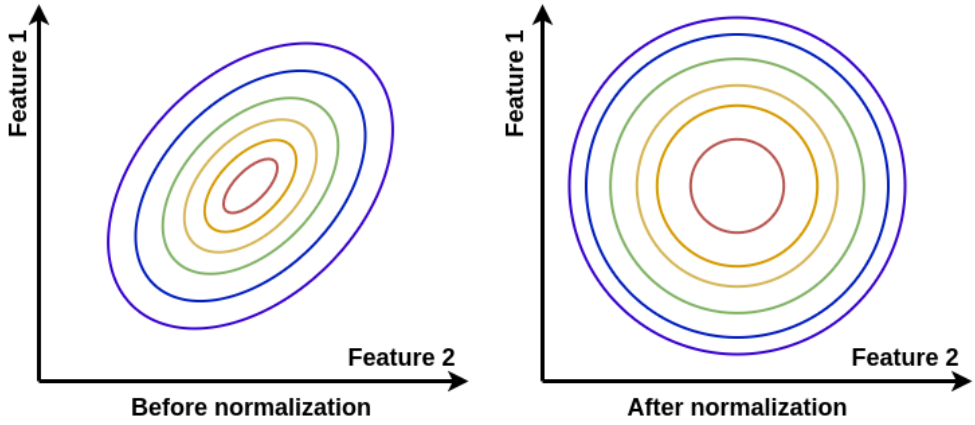

Figure 11. Features are normalized into the same scale.

Batch normalization uses four parameters in each batch normalization layer. Batch variance (σ^2) and batch mean (μ) are used for shifting and scaling of variables. Other parameters are standard deviation (γ) and mean (β) parameters, which are learned during the normalization process. [23]

Parameter learning in a batch normalization is based on Equations (2) and (3), where $\boldsymbol{b} = \{x_1, \ldots, x_m\}$ is the input batch.

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2 \tag{3}$$

Each normalization layer applies Equation (4), where x is an input element of the layer, x' is the normalized element value, and ϵ is a small constant for numerical stability. [23]

$$x' = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{4}$$

With these equations, the process enables more independent learning of parameters for each layer. σ and μ are sometimes randomly initialized, as previous optimizations might no longer be optimal: an optimizer might unlearn these normalizations if it is a convenient way to minimize the loss function.

γ and β parameters in a batch normalization transform change weights per activation in Equation (5), where x is the input to a layer and x' is the normalized value. γ and β are used for batch scaling and shifting. [23]

$$x' = \gamma x + \beta \tag{5}$$

Batch normalization reduces internal covariate shift and the dependence of gradients on the scale of parameters, regularizing the model and normalizing layer level responses [23]. This is an important aspect in quantization, which saves energy, improves computing speed, and enables higher learning rates. [24] For inference, the network needs to do the same normalizations as during training. These normalizations can be calculated offline by folding [25].

2.2.2. Convolutions

Let's presume we have an input signal $x(n)$, an impulse response $h(n)$ and an output signal $y(n)$. Equation (6) is a convolutional system, where $\circledast$ is the convolution operator. [26]

$$y(n) = x(n) \circledast h(n) \tag{6}$$

In CNNs convolutional layer computations essentially extract features. In Equation (7) the output vector ($\boldsymbol{y}$) is equated, where $\boldsymbol{W}$ is the weight matrix, $\boldsymbol{x}$ is the input vector, and $\boldsymbol{b}$ is the bias vector.

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{7}$$

$\boldsymbol{W}$ is represented as a multi-dimensional array and $\boldsymbol{b}$ is defined as an vector in Equation (8).

$$\boldsymbol{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \dots & \dots & \dots & \dots \\ w_{m,0} & w_{m,1} & \dots & w_{m,n} \end{bmatrix} \boldsymbol{b} = \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_m \end{bmatrix} \tag{8}$$

Convolutional layers are fundamentally built on local receptive fields, shared weights, and spatial sub-sampling. Local receptive fields refer to design architecture, where convolutions are connected to their input layer. Input image can contain millions of pixels, but we can detect meaningful features, that only inhabit a region of dozens of pixels, as the convolution kernels are small.

The execution time depends on the kernel size [26]. Figure 12 visualizes convolutional layers and the kernel computation flow from layer to layer.

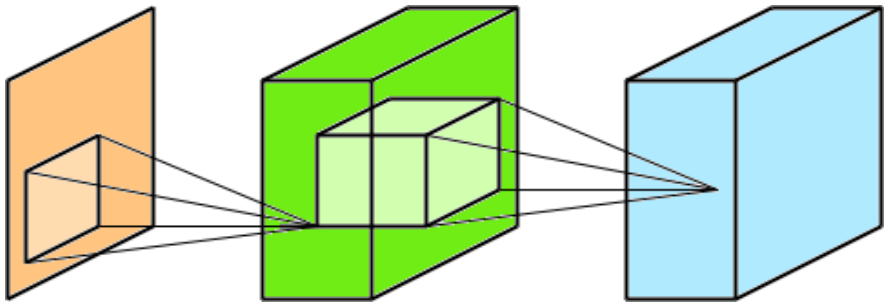

Figure 12. Convolutional layers form a chain of operations in neural network.

2.3. Neural Network Optimization

Neural computing acceleration needs to consider operator throughput, memory access compression, precision arithmetic, bitwidth, and quantization. The aim is to accelerate inference, to reduce model size, and to achieve better energy efficiency. The problem with most methods is the inability to exploit sparsity or value homogeneity. [25]

It is known that classification neural networks are robust under heavy quantization noise, and 8-bit, or even 4-bit precision may suffice [25]. Among the optimization techniques are weight clustering, quantization aware training (QAT), post-training quantization (PTQ) and pruning. The most effective optimization method is QAT, but it is the hardest to deploy. PTQ is relevant technique in many cases, as the accuracy difference between QAT and PTQ is usually marginal [27].

2.3.1. Quantization

In this thesis quantization means mapping 32-bit floating point models into lower precision and numeric range. Important attributes in quantization are dynamic range and precision. Dynamic range refers to the range of variable space and precision is the number of levels in range [25]. Range and precision impact the amount of quantization error.

Quantization of a signal is visualized in Figure 13, where quantization error is the rounding error between the original and quantized signals, and the number of quantization levels is defined by the value range of original signal.

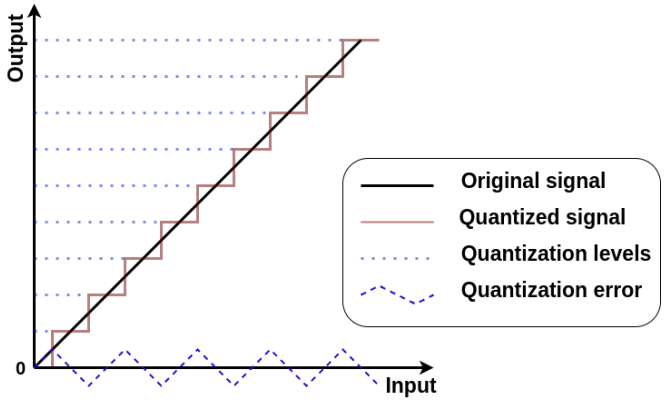

Figure 13. Visualization of signal quantization.

Most common quantization approaches in convolutional neural computing are range- and clipping-based approaches. Range-based quantization approaches use the full variable range. Clipping-based approaches use a static or a learned range that results in limiting the maximum and minimum values. Common quantization methods are symmetric and asymmetric methods. [28]

Typically three variables are used in quantization:

- Common exponent

  The exponent is often called zero-point. It is the offset of values, and is a neural computing specific constraint. For many operations we need the real value of 0 to be exactly representable in a quantized form, as it is an optimality constraint. [24]

- Scaling factor

  The scaling factor is used to adapt weights of tensors to a dynamic range. Scaling factor is usually defined per-channel or per-layer. [24]

- Minimum and maximum activation ranges

  The range functions are applied to the variable before and after to-be quantized layers and they map the activation range of each layer. [24]

Quantization of a scalar is defined in Equation (9), where $\boldsymbol{X}_{ij}$ is the input element, $\boldsymbol{X}_{q(ij)}$ is the quantized value of x, z is the zero-point, and s is the scaling factor. [24]

$$\boldsymbol{X}_{ij} = (\boldsymbol{X}_{q(ij)} - z) * s \tag{9}$$

Digital signal processors (DSPs) and tensor processing units (TPUs) used in neural computing rely on quantized inputs. For eliminating as much floating point arithmetic as possible, the matrix multiplications (MMs) must be quantized.

Computation using a low precision general matrix multiplication (GEMMLOWP) algorithm is in Equation (10), where notations $\boldsymbol{R_{q(ij)}}$ are the quantized values, R_z the zero-point, and R_s the scale of the result, and $\boldsymbol{LHS_{q(ij)}}$ and $\boldsymbol{RHS_{q(ij)}}$ are quantized left-hand and right-hand side matrices. [29]

$$\boldsymbol{R}_{q(ij)} = R_z + \frac{lhs_s * rhs_s}{R_s} * ((\boldsymbol{LHS}_{q(ij)} - lhs_z) * (\boldsymbol{RHS}_{q(ij)} - rhs_z)) \tag{10}$$

The innermost part of an accumulation is called the kernel loop. In neural networks it is the most computationally intensive part. Elimination of as many variables as possible from the kernel loop reduces the computation cost of MMs. Neural network inference is often measured in terms of multiply accumulate operations (MACs). Accumulator size is minimized as computational cost significantly increases along the size of the accumulator. [29]

Optimal accumulation A_q for a quantized general matrix multiplication is defined in Equation (11).

$$\boldsymbol{A}_{q(ij)} = \boldsymbol{LHS}_{q(ij)} * \boldsymbol{RHS}_{q(ij)} \tag{11}$$

Post-training quantization is a common technique used to optimize model inference. In parameter abundant networks there are outliers in learned weights, so losing information on these variables does not change the model behaviour fundamentally [24]. Smaller models are different as their lightweight architecture provides smaller representation capacity [30].

2.3.2. Folding

CNN computation is a repetitive pipeline, which consists of calculation, normalization and activation of features. Figure 14 visualizes the most repetitive neural computation scheme, a block, which consists of convolution, batch normalization and activation layers.

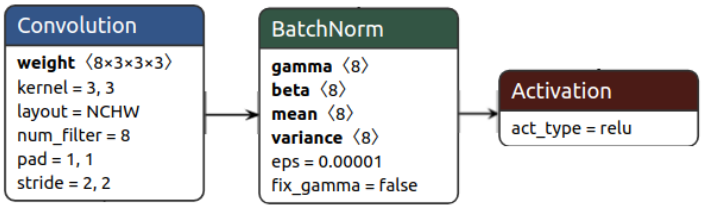

Figure 14. Convolutional, batch normalization and activation layers.

Folding is a technique where offline calculable weights are fused to weights of another layer. This decreases the number of operations per inference. Usual way for folding is fusing together convolution, normalization and activation layers. [24]

Folding process is visualized in Figure 15 where input, convolution, and activation weights are quantized. Quantization nodes are used for quantization, and they contain information about the minimum and maximum activation ranges of the layer. [24]

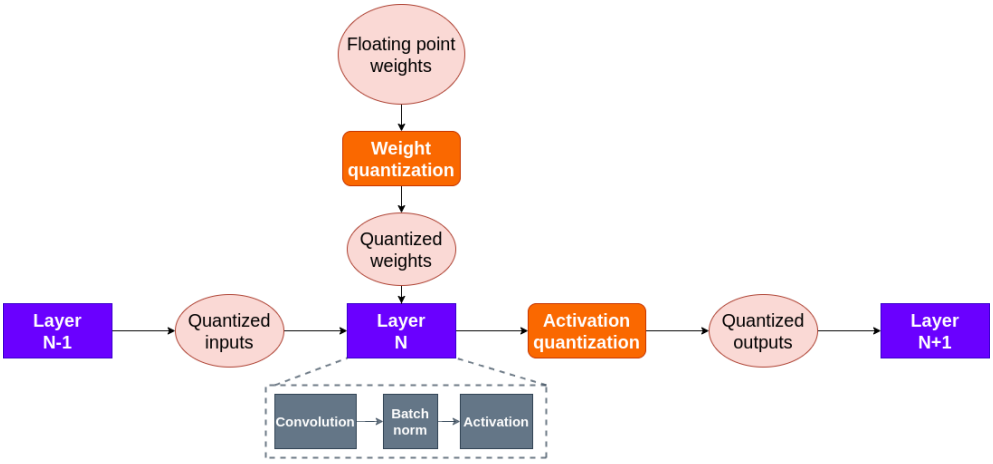

Figure 15. Folding of convolution, normalization and activation layers.

The convolution layers consist of weights ($\boldsymbol{W}$) and biases ($\boldsymbol{b}$) [24]. Folding of a weight w, which is one of the elements defined in Equation (8), is presented in Equation (12), where γ is batch standard deviation parameter, σ^2 a variance parameter and ϵ a small constant for numerical stability. [23]

$$w_{fold} = \frac{\gamma * w}{\sqrt{\sigma^2 + \epsilon}} \tag{12}$$

Folding of bias element b, which is defined in Equation (8), is presented in Equation (13), where β is the mean parameter of batch normalization. [23]

$$b_{fold} = \frac{\gamma * (b - \mu)}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{13}$$

Equations (12) and (13) are combined in Equation (14). With this folding technique both weights and biases of a convolutional layer can be folded. [23]

$$x_{fold} = \frac{\gamma * w(b - \mu)}{\sqrt{\sigma_w^2 + \epsilon}} + \beta \tag{14}$$

An example of the quantization and folding of a block is presented in Appendix 1.

2.3.3. Pruning

Pruning is a network compression technique. Pruning is based on the fact that model has many weights that hold little to no information for the detection [31]. Network sparsity is the goal of pruning. Layer sparsity is defined in Equation (15).

$$|\boldsymbol{X_p}| = \begin{bmatrix} |x^0_{0,0}| & |x^0_{0,1}| & \dots & |x^0_{0,n}| \\ |x^0_{1,0}| & |x^0_{1,1}| & \dots & |x^0_{1,n}| \\ \dots & \dots & \dots & \dots \\ |x^0_{m,0}| & |x^0_{m,1}| & \dots & |x^0_{m,n}| \end{bmatrix} \tag{15}$$

Sparsity can be conducted on multiple levels: on fine-grained-, vector-, kernel-, and filter-level. Better results can be achieved by using smoother pruning algorithms [32], but there are no general guidelines what is the breaking point of a model [33]. In Equation (16) calculation of the zero-element count of a layer takes place.

$$x = \sum_{i=0}^{n} \sum_{j=0}^{m} |x_{ij}^{0}| \tag{16}$$

Neurons with no connections can be safely pruned in any pruning algorithm. The most straightforward pruning technique is one-shot pruning, where algorithm randomly deletes data based on threshold. The more advanced pruning includes training-aware pruning, where a training optimizer is used for fine-tuning. The idea is to learn which weights are the least important. [33]

Traditional neural architectures are error tolerant by design, but mobile neural architectures are not as error tolerant due to smaller parameter count [31]. However, for most classification networks, pruning of later layers should prove useful as long as the most critical layers are avoided [34].

2.3.4. Clustering

Clustering is based on weight sharing. Weight sharing means dropping the number of unique weight values in a model. Weights are put into clusters and each weight belonging to the cluster is replaced with centroid value of the cluster. K-means implementation on neural computing is illustrated in Figure 16. [35]

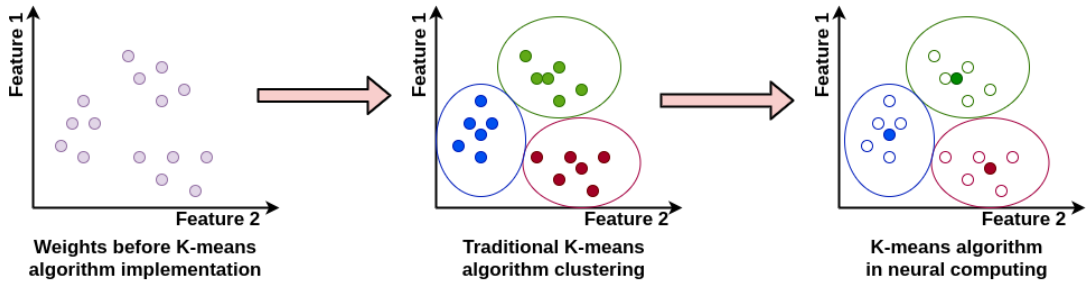

Figure 16. Feature homogeneity is increased by use of clustering methods.

2.4. Summary

Normal computations can be accelerated and the energy efficiency improved by quantization, and by exploitation of the specific opportunities provided by the ReLU activation function. By folding offline calculable weights the number of operations can be reduced. By applying sparsity to the model the model size can be reduced, but pruning has narrow applicability on mobile networks as the number of parameters is limited. In mobile implementations, high level performance is desired while the computational solution should be minimized.

3. NEURAL COMPUTING ON EMBEDDED PLATFORMS

Understanding hardware and model design is fundamental in training and inference optimization. Most optimization methods concern one or multiple aspects of throughput, efficiency, latency, accuracy and memory usage. Usually optimization is a trade-off between these issues. [36]

The most vital components used in neural network training are memory, central processing units (CPUs), accelerators (e.g. GPUs), high-speed connectors (network connection and buses) and fast storage. Seamless design of the computing pipeline is a requirement for producing a low latency solution.

3.1. Number Representations

A computer representation is always restricted due to storage requirements. There are two main types of numbers modern microprocessors operate on: fixed- (FX) and floating point (FP) representations [37]. Fixed-point (FX) numbers represent fractional values and are heavily used in low-cost embedded microprocessors. Figure 17 illustrates FX 8-bit (FX-8) unsigned (uint8) and signed (int8) representation examples, where s is a sign, $\boldsymbol{i} = \{i_1, \ldots, i_m\}$ is the integer part, and $\boldsymbol{f} = \{f_1, \ldots, f_m\}$ is the fractional part of a representation [38].

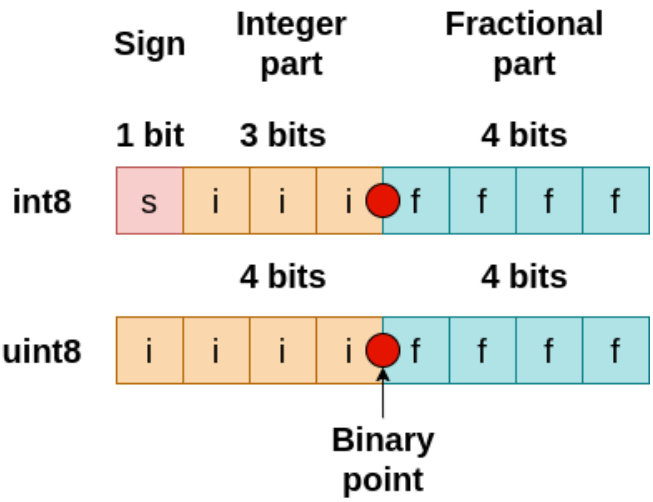

Figure 17. Representations of important signed and unsigned FX formats.

Neural networks are often trained using FP arithmetic, but for efficient neural network inference we need to map FP to FX representation. For mapping a FX presentation from a FP presentation, we need to consider the dynamic range of the conversion, and precision. Dynamic range is defined by the largest and smallest numbers, and precision is the number of levels in the representation. FX arithmetic is often represented in Q-, or UQ-format, for signed and unsigned, respectively, where m is the integer bit count and n is the fractional bit count. [26]

FP to FX optimization can improve performance and accuracy. In used optimization methods the FX scaling factor is shared. [36]

FX approximation of a FP number can be calculated by Equation (17), which is the two's complement representation. [38]

$$FX = \lfloor (FP * 2^m) \rfloor \tag{17}$$

The required FX representation for optimal neural 8-bit precision is (U)Q1.7 format.

FP arithmetic is the most common way of representing numbers in computing systems. FP consists of three parts: The sign s, the exponent $\boldsymbol{e} = \{e_1, \ldots, e_m\}$ and the mantissa $\boldsymbol{m} = \{m_1, \ldots, m_m\}$ [39]. FP arithmetics are visualized in Figure 18. In a FP system the compromise is made between the mantissa and the size of the exponent [40]. FP formats useful for neural computing are:

- IEEE 754 floating point 32-bit (Single-precision) representation (FP-32)

  Single-precision FP is typically used during training of neural network.

- IEEE 754 floating point 16-bit (Half-precision) representation (FP-16)

  Mapping between IEEE 754 FP data types requires conversion of values due to differences in exponent and fractional part sizes.

- Brain floating point 16-bit representation (BFP-16)

  Google brain team has presented brain floating point format (BFP-16) for usage instead of conventional FP-16. BFP-16 uses same amount of bits in exponent part making the conversion easier to implement. The range of BFP-16 is the same as with FP-32. The hypothesis is to have better classification accuracy on BFP-16 compared to FP-16. [41]

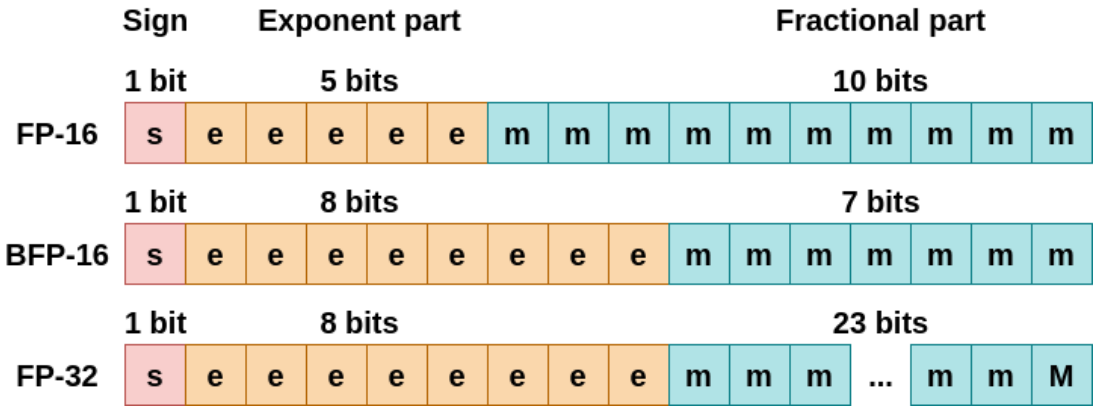

Figure 18. Representations of important FP formats.

In hardware, saturated or unsaturated arithmetics are used. Saturated arithmetics clip over- or underflow to maxima or minima, respectively. Unsaturated arithmetics allow overflows. [42]

3.1.1. Number Formats for Neural Networks

CNNs employ FP-32s for storing gradient weights. This is currently the most commonly used procedure for keeping gradient values from vanishing during training [17]. In inference we need only to compute the forward pass. Typically FP-16 or FX-8 suffices for classification networks [25]. When utilizing a particular arithmetic, speed, accuracy, range, portability, ease of use, and speed are the core attributes [38].

3.2. Computing Concurrency

Computing concurrency in neural computing has many levels, most relevant ones for inference being operator and network concurrency. At operator level there are two scales: the single operator and the decomposition of operators. Decomposition of operators deals with convolutions. [43]

Data parallelism deals with batch size and communication, which are important for training of a network, but model parallelism and pipelining are relevant for latency optimization. Model parallelism deals with layer level optimization methods and memory footprint reduction. Pipelining concerns the data flow of a model and layer process partitioning. Hybrid parallelism is a combination of multiple parallelism schemes, it is a common practice to combine multiple parallelism schemes. [43]

Multiprocessing and parallel processing, in theory, provide for linear speedup, while Amdahl's law shows it is limited by serial parts of the application [44]. In practise linear speedup is limited due to communication and memory bottlenecks. Load balancing, communication, and processor synchronization are important aspects for effective parallelization [45]. Different computing schemes are visualized in Figure 19, where $P = \{1, 2\}$ are processors.

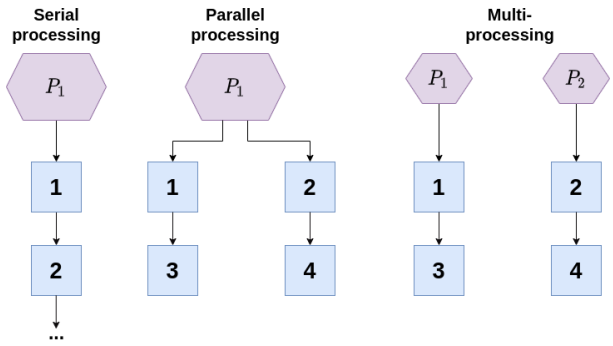

Figure 19. Different processing schemes.

3.2.1. Convolution Algorithms

One of the most important hardware features for neural computing is the effectiveness of convolution computation. Convolutional layers in neural computing are the SOTA method for feature representation, but are heavy to compute due to high number of operations. Convolutional layers include majority of computations involved in

inference and training of CNNs [43]. Convolution optimization algorithms reduce the total number of operations and communication [46].

Three convolution algorithms are typically used in neural computing:

- im2col

  In im2col a discrete convolution is transformed into a matrix multiplication (MM). The im2col method is good method for acceleration, but it consumes a high amount of memory which results in bad scalability. im2col is also known as general matrix multiplication (GEMM) method. [43]

- Fast Fourier Transform (FFT)

  Compared to im2col, FFT algorithm is efficient for large convolutions due to weight reuse. FFT can be optimized further by usage of zero-padding due to many kernel values being zero. For optimality pruned FFT is especially efficient. In pruned FFT number of operations is reduced. [43]

- Winograd's fast convolution algorithm

  Winograd's algorithm cuts the number of multiplications, but its cost grows quadratically with kernel size so it is only used for small kernels. Winograd's algorithm affects the numerical accuracy. [47]

Visualization of each algorithm is in Figure 20, which is adapted from [43] and [47]. In im2col 3D tensors are disassembled into 2D matrices. Kernels and image data are multiplied and the result is the convolved tensor. In FFT algorithm data and kernels are transformed by FFT and element-wise multiplied. Inverse FFT is then applied for getting the result. In Winograd's algorithm, convolutions are decomposed into sums of tiled small convolutions from transformed kernel and activation layer products. [43]

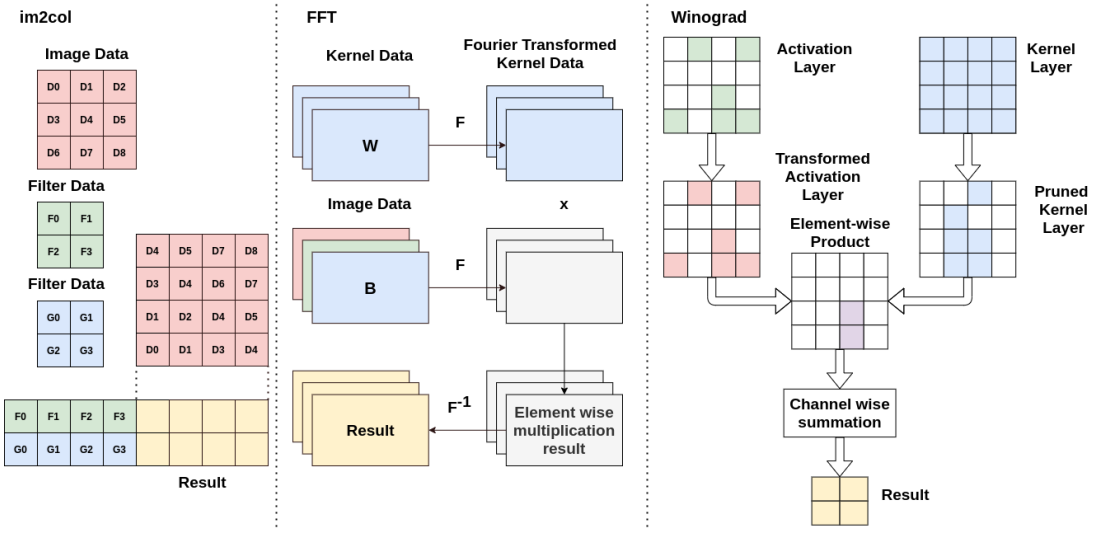

Figure 20. Convolutional Computation method visualizations.

There is no effective general use algorithm for neural convolution computation. The layout of the data has a huge effect on acceleration speedup of each convolution optimization algorithm. [43]

3.2.2. Data Order

For a high-performance matrix computation system, minimization of operation count is not enough. Data order has a significant role for the performance. All inputs, outputs and transformations of the data are represented in tensors that are multidimensional arrays [43]. In Equation (18) an example of input data order for a tensor is visualized, where $\boldsymbol{T}$ is a tensor, $\boldsymbol{B}$ is the batch size, $\boldsymbol{C}$ is the number of channels, $\boldsymbol{H}$ is the height and $\boldsymbol{W}$ is the width of a tensor [36].

$$\boldsymbol{T} = [\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{H}, \boldsymbol{W}] \tag{18}$$

Convolution processing efficiency is related to the processing architecture. Most commonly the [$\boldsymbol{C}$,$\boldsymbol{H}$,$\boldsymbol{W}$,$\boldsymbol{B}$] format is used, which is the interleaved format. [48]

3.2.3. Pipelining

FP operations require multiple steps for finishing a calculation, as a clock cycle is used for each step. When we feed x and y to an adder we get an output sum z. Figure 21 represents example figure of a FP adder, that requires three clock cycles. [49]

Figure 21. A 3-cycle adder.

Pipelining means streaming each phase at one cycle. We can expect thrice better processing throughput by using pipelined addition [49].

For neural computing concurrency acceleration it is limited by the longest-latency element in the pipeline. Pipelining can be conducted on layer and model levels. Pipelining in neural computing needs to address data dependency and resource allocation related challenges. [43]

During inference the data is used when available, but data dependency is a bottleneck when using batch normalization techniques. Layer partitioning has been shown to reduce the required amount of parameters and communication between processors [43]. Figure 22 visualizes pipelined execution, where $\{1, \ldots, 4\}$ are execution groups.

For efficient pipelining specialized hardware, such as systolic arrays, are used [50].

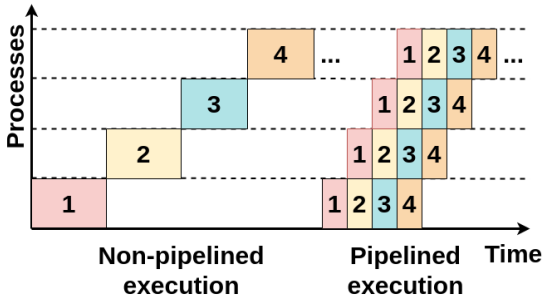

Figure 22. Pipelined computation visualization.

FX calculations are simpler, however both FX and FP representations are restricted in precision and dynamic range. Restrictions manifest rounding and saturation errors, which propagate through networks, and the effects on the networks are hard to analyze, requiring experimentation [51]. Design of a concurrent system requires consideration of instruction and data stream(s). A popular taxonomy for computation concurrency is Flynn's taxonomy [52].

3.2.4. Instruction and Data Streams

An instruction stream is an sequence of instructions performed by the machine and a data stream is a sequence of data called by the instruction stream. A program is an ordered set of instructions. [53]

Usage of multiple threads is called multithreading (MT). MT is a prominent way of accelerating computing. In interleaved MT the tasks are coarse- or fine-grained, where thread executions are based on cycle counts, cache misses and fairness. [43]

Figure 23 displays the idea of fine-grained parallel execution, where threads are used for asynchronous tasks $\boldsymbol{t} = \{1, \ldots, 8\}$ processing.

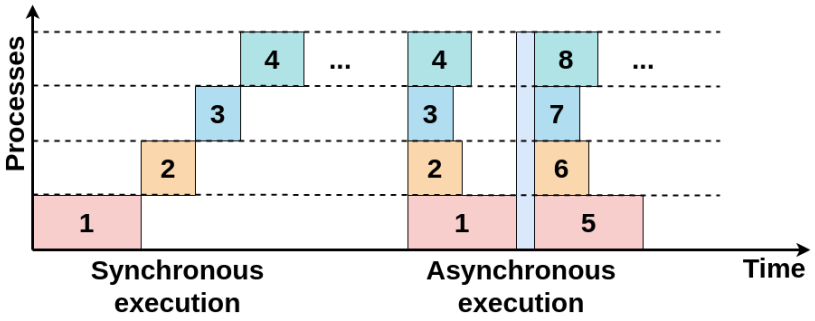

Figure 23. Multiple threads are used for accelerating computational tasks in fine-grained parallelism.

The weaknesses of fine-grained parallelism appear when dealing with elements that need to be saved to cache for long time spans, while coarse-grained tasks are good for environments where large number of cache misses are an issue. [43]

Single instruction, multiple threads (SIMT) and multiple instructions, multiple threads (MIMT) architectures are used for efficient neural computing. In neural computing

- Single instruction, multiple threads (SIMT)

  Single instruction, multiple threads (SIMT) is based on idea of thread warping, where warps are collections of threads. Multiple entities of data are accesses simultaneously and divided into smaller tasks [54]. Advantages of using the SIMT model are thread individuality and flexibility of warping, which translate to inference acceleration possibilities by fine-grained task scheduling.

  Figure 24 illustrates differences between Single Instruction architecture pipelines.

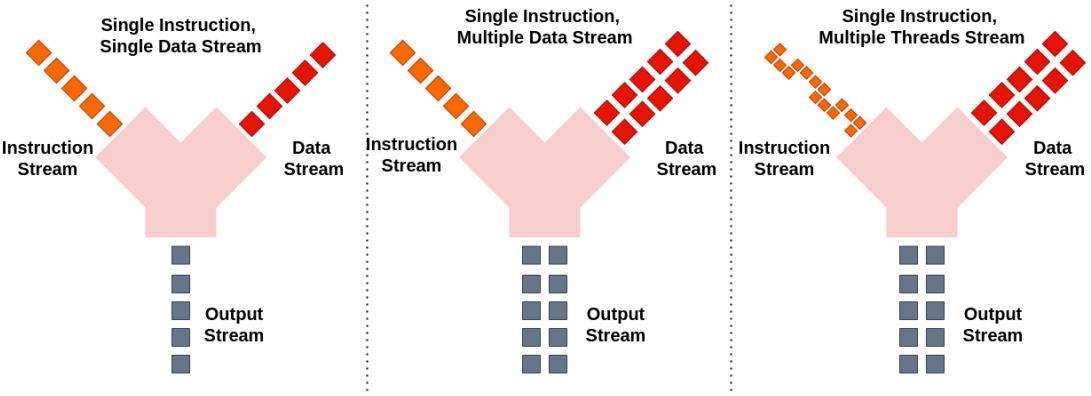

Figure 24. Single instruction stream architectures.

- Multiple instructions, multiple threads (MIMT)

  MIMT is an execution model used in parallel computing, where system utilizes multithreading with multiprocessing. In multithreaded computing a set of parallel threads are grouped into warps. Ideal warp execution happens without conflicts. Different conflict cases are instruction and data level read misses, and data write misses. If threads of a warp come to a data dependent branch, the warp serially executes each branch till the paths are completed. [54]

  The inference acceleration idea behind MIMT is based on fine-grained thread warping. Instruction level warping for multiply-add is in Figure 25.

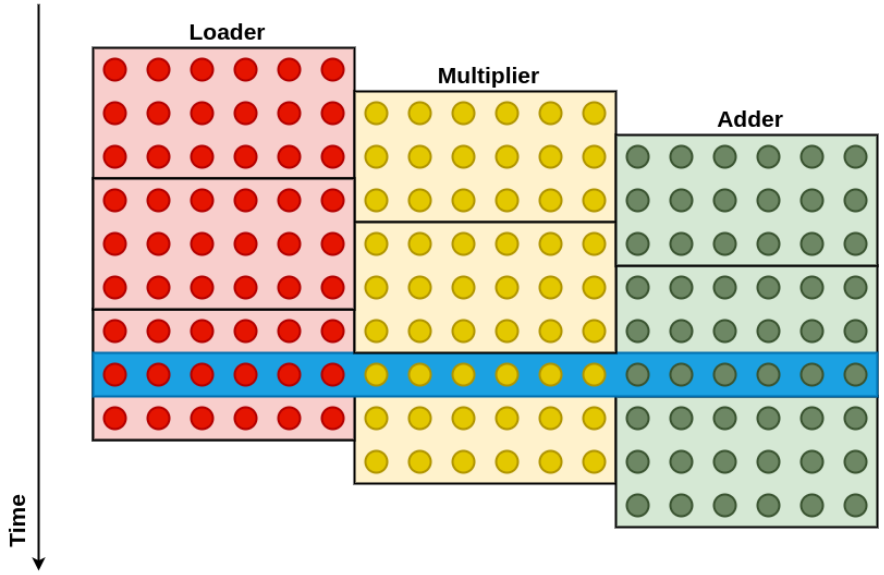

Figure 25. Optimal instruction level parallelism in multiply-add.

Multiple instruction architectures offer a possibility for simplifying matrix multiplication (MM) algorithms, but simplifying the algorithms is not straightforward process as systems have different bottlenecks and data dependencies. [46]

3.3. Delegation

CPU, GPU and DSP are exhibits of instruction set architecture (ISA) implementations. An implementation is built on top of instructions. A system consisting of multiple processors with different architectures is called a heterogeneous computer. Heterogeneous computing is typical when reaching for an optimized computing system, where computing is delegated to the most efficient platform. [50]

Data handling, memory operations and control flow operations are cornerstones of ISA design, where parallelism is divided into three types:

- Task-level

  Task-level parallelism is achieved when different threads are executed on different or same data.

- Data-level

  Data-level parallelism is achieved when each processor performs same task on different pieces of distributed data. Location of memory access affects read and write times and cause of increasing memory latency. This is one reason why memories are grouped into private and shared memories. The smaller the memory is in size, the faster the processing speed. [45]

- Instruction-level

  Instruction-level parallelism is achieved when instructions are grouped and executed in parallel without changing the end result. Instructions can only be grouped together if there is no data dependency between instructions. [45]

CPUs commonly use SIMT architecture, but are optimized for serial operations. CPUs have large cache sizes and complex control logic, but low processing unit capability. Many computation pipelines map data initially to CPU accessible memories, which account to low initialization durations.

The most common delegation platforms for smartphones are GPUs and digital signal processors (DSPs). Less common mobile delegation platforms are field programmable gate arrays (FPGAs). For delegation the system needs to decrease information richness, as the information processing pipeline is dependent on used delegation hardware and software.

3.3.1. Graphics Processing Units

GPUs have high memory access costs. It is therefore important that the processing time is optimized against data transfers. Boosting performance of a GPU is done by keeping memory access patterns local. [45]

GPUs used for neural acceleration excel either at training or inferencing. Biggest problems with GPU delegation are enormous memory allocation times, which is the critical factor in mobile GPU delegation. GPU architectures specialize for maximal FP burst reading throughput, but are not good at handling interrupts and sparse data formats,. The way of boosting GPU performance is to keep memory access patterns

local and to minimize handling interrupts and sparsity of data. Generally data transfer is the most common bottleneck when delegating on a GPU. [55]

3.3.2. Digital Signal Processors

Typical MIMT implementations are parallel DSPs used in digital image processing. Generally main benefits are lower power consumption and better inference speed compared to CPUs and GPUs. In our context DSPs employ FX inference pipeline.

Fundamentally DSPs are designed to perform some basic operation very quickly. System should be designed to be sustainable and meet minimum requirement for task processing speed as processing speed increases the power consumption. Power consumption of algorithm is affected by the complexity of the algorithm. [56]

3.3.3. Field Programmable Gate Arrays

Field programmable gate arrays (FPGAs) provide high speedups and data parallelism compared to regular processors. FPGAs are reprogrammable special-purpose hardware. Relevant design techniques for accelerating MM algorithms on FPGAs are systolic arrays. [57]

3.3.4. Systolic Arrays

Systolic systems consist of cells, which are processing- (PUs) and memory units. Systolic design is bound on a particular computational task. The systolic array is a good design choice memory bandwidth-, computational output- and modularity-wise as the system is simple and regular by design. [54]

The idea of a systolic system is to reduce latency by reducing the communication time and by having a pipeline of operations, enabling processing of multitudes of operations compared to the original system. Figure 26 represents a systolic system, where the processing units form a chain of operations.

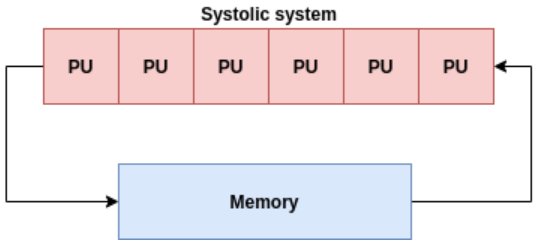

Figure 26. A systolic system is a set of processing units working together for solving a task.

Producing special systolic hardware comes with a price: for a large input size the cost and performance increase proportionally, but due to usage of simple elements the design will remain regular: it needs to be tailored to the problem. [54]

3.3.5. Tensor Processing Units

For efficient tensor processing unit (TPU) design, a matrix multiplier unit (MXU) is one of the most specialized designs. TPU is a specialized architecture for NN inference, where MXU consists of FX-8 multiply-and-add operations. MXU is a matrix processor, which is capable of processing hundreds of thousands of matrix operations in a single clock cycle. [58] The MXU provides vast number of multiplies per second and usage of variables without need for intermediate storage. The weight pipeline forms a partial sum pipeline. A MXU is visualized in Figure 27, where cells form a systolic array for multiplications.

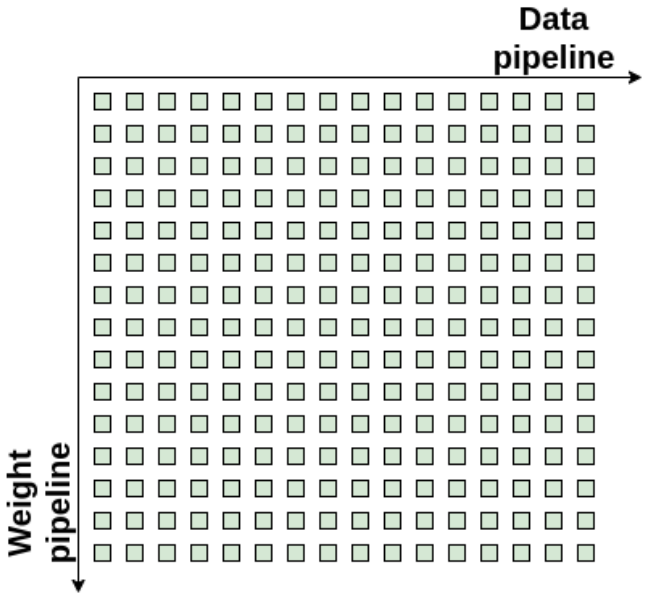

Figure 27. MXU flow in a TPU.

A systolic array is used for allowing the data to flow into the matrix multiplication without storing variables to registers. While TPUs offer accelerated performance, they are single purpose application-specific integrated circuit (ASIC) designs. [59]

3.4. Summary

Computation can be reduced by mapping the variables into smaller number formats. The representations are important factors for enabling a trade-off for precision and inference acceleration of a neural computation system. Pipelining and parallelism are key elements which have led to many coherent concurrent hardware implementation designs. The general convolution algorithm can be simplified by replacing it with faster convolution methods. Neural inference can be accelerated by dividing computational tasks into fine-grained threads, which are coordinated between processing cores. Computation throughput can be accelerated by using special hardware with narrow implementation pipelines, but bottlenecks for each hardware vary.

4. FACE DETECTION IMPLEMENTATION

The Common face detection pipeline consists of three steps. The first step is filling the input, the second is inference and the third step is output decoding. The objective on the last step is turning output to format where it becomes useful information for the end user. In our implementation RetinaFace project was chosen for the baseline. The reasons being clear technical documentation and the results on public benchmarks. [4]

4.1. Input Fill

Initially the input image is transformed to the needed format and moved to memory for the inference engine.In TensorFlow Lite the input tensor expects the model format to be formatted in $[B, H, W, C]$ format. In Figure 28 the input image is transformed from planar to interleaved image representation.

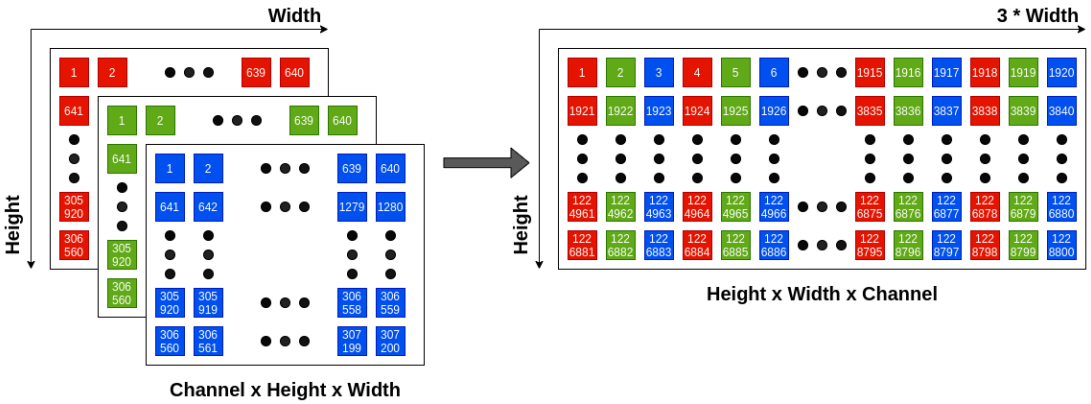

Figure 28. Converting image from continuous to interleaved format.

4.2. RetinaFace Computing

Before the model inference, the input dimensions are allocated, and the model is initialized. The inference engine calculates weights for the input image. The forward pass is used for producing outputs. Outputs need to be decoded after extraction.

The most common decoding approach is using bounding boxes and classification scores. Classification scores are used as the first discarding value. For bounding box representation, in a single shot detector scheme, we need to connect anchor boxes to the matching bounding boxes. Bounding box voting is used to discard non-interesting predictions. Bounding box voting consists of intersect over union (IoU) [60], and non-maximum suppression (NMS) [61] techniques. Single shot detection phases are visualized in Figure 29.

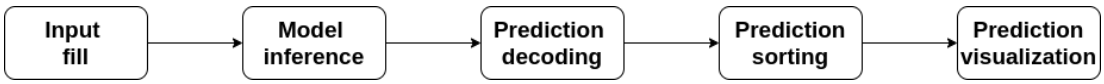

Figure 29. The Single shot detector pipeline.

4.3. Model Outputs

Outputs of the model are prediction scores, and bounding box deltas. Prediction scores are extracted from softmax layers, and bounding box deltas are extracted from convolutional layers. The outputs are designed to be extracted for each anchor per the densification parameter. Densification means the number of bounding box locations for each anchor position.

4.3.1. Prediction Scores

Output level always contains an output layer for extracted bounding boxes and classification scores. Prediction scores need to be mapped according to the transformations applied to the bounding box outputs (bounding box deltas).

4.3.2. Bounding Boxes

We want to extract minimum bounding boxes for detected faces. Bounding boxes adopt scale-invariant transformations for the centers, and log-scale transformation for the height and width. For determining a prediction position, the prediction center position, and the bounding box delta and anchor are used, which are visualized in Figure 30, where offsets are the extracted delta variables of a bounding box layer.

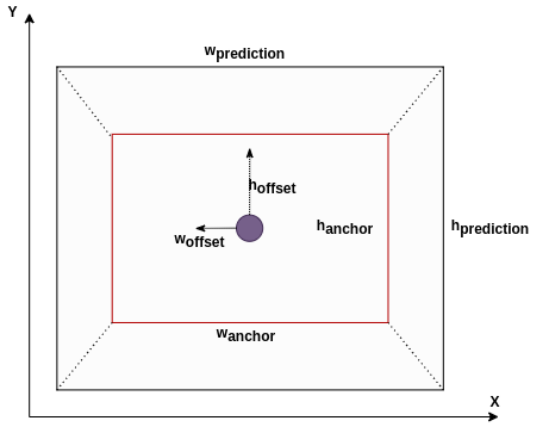

Figure 30. The bounding box decoding phase.

A minimum bounding box is visualized in Figure 31. The detection is defined to tightly contain the forehead, chin and cheek. If the face is occluded, parameters are used for placing the bounding box on the estimated location of the face. [3]

Figure 31. An example of minimized bounding box.

4.3.3. Anchors

After we have decoded the bounding box deltas the generation of anchors commences. An anchor is an default position for a bounding box. The face detector implementation contains three levels. The anchors are calculated based on the levels and each level defines a stride length. Strides are fixed constants with lengths of 8, 16 and 32 pixels.

4.4. Predictions

We have a number of predictions by linking the bounding box predictions with matching scores. By reshaping the tensor data into shape where we can easily align corresponding bounding boxes with prediction scores. An example of predictions with high detection scores are visualized in Figure 32, which consists of example images from [3]. A voting scheme is needed for removing predictions representing the same faces.

Figure 32. Examples of predictions with high confidence scores.

The previous processes generate a large number of detections. The detector needs to determine acceptable discarding criteria for predictions. Bounding box voting is a set of techniques, where the aim is to limit proposed detections per object to the most matching detection. Intersect over union (IoU) and non-maximum suppression (NMS) are used for voting of predictions.

4.4.1. Intersect over Union

In intersect over union (IoU) the area of each bounding box is measured and the intersection between each targeted bounding box is calculated. Figure 32 shows examples of detections without applying voting techniques. An example visualization of IoU is in Figure 33.

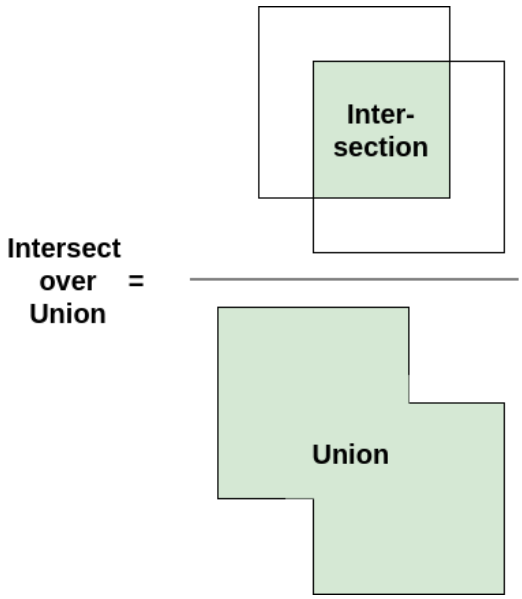

Figure 33. Intersect over Union visualization.

IoU is a popular evaluation metric used in face detection for removing unwanted candidate predictions. For calculating IoU a cost function is used, common ones being Manhattan and Euclidean distances. The formulae of IoU is in Equation (19), where a and b are the areas of predicted bounding boxes, i is the intersection area and u is the union area.

$$IoU = \frac{|a \cap b|}{|a \cup b|} = \frac{i}{u} \tag{19}$$

Suggested alternative for IoU is the generalized IoU (GIoU). If the voted predictions do not overlap, the value of IoU is zero. GIoU is defined in Equation (20), where c is the smallest convex area that encloses both predictions. The smaller c value, the better prediction score is. [60]

$$GIoU = IoU - \frac{|c/(a \cup b)|}{|c|} \tag{20}$$

4.4.2. Non-Maximum Suppression

NMS is a post-processing technique where a set of bounding boxes are transformed into a single detection. The most important parameter of the voting scheme is the threshold which is used for discarding boxes with large IoU score.

In our implementation we use a greedy-NMS algoritm. Greediness means the system assumes the best scoring bounding box covers the object and drops other bounding boxes based on the suppression threshold.

When each bounding box below threshold value is removed, the second highest scoring window is selected and the procedure is repeated until all windows have been iterated through. Combining the IoU and NMS gives Algorithm 1.

Algorithm 1. Bounding Box voting scheme

In voting p stands for prediction, t stands for target and i stands for intersect. Before this algorithm we have sorted predictions from ascending by prediction.

for $p\ preds$ **do**

  $threshold = 0{,}5$

  $p_{a_{rea}} = (p_{x_{max}} - p_{x_{min}} + 1) * (p_{y_{max}} - p_{y_{min}} + 1)$

  for t **to** $preds$ **do**

    $i_{x_{min}} = max(p_{x_{min}} - t_{x_{min}} + 1)$
    $i_{y_{min}} = max(p_{y_{min}} - t_{y_{min}} + 1)$
    $i_{x_{max}} = max(p_{x_{max}} - t_{x_{max}} + 1)$
    $i_{y_{max}} = max(p_{y_{max}} - t_{y_{max}} + 1)$

    $i_{a_{rea}} = max(i_{x_{max}} - i_{x_{min}} + 1) * max(i_{y_{max}} - i_{y_{min}} + 1)$
    $t_{a_{rea}} = (t_{x_{max}} - t_{x_{min}} + 1) * (t_{y_{max}} - t_{y_{min}} + 1)$
    $iou = i_{a_{rea}}/(p_{a_{rea}} + t_{a_{rea}} - i_{a_{rea}})$

    if $iou > threshold$ **then**
      erase t from $preds$
    else
      continue
    end
  end
end

After processing the detection proposals, the detector outputs the selected outcome. In optimal cases the detection contains one detection per object in an input image. In Figure 34 we have removed the detections representing the same object. Notice the difference to Figure 32. Images taken from [3].

Figure 34. Examples of voted predictions.

4.5. Implementation Toolkit

The detector is implemented using TensorFlow (TF) C++ API and TensorFlow Lite (TFLite). [59]

4.5.1. TensorFlow

TF is a framework for neural network computing. It supports both model training and inference. The design of a neural network is done by using dataflow graphs. Important aspects, why TF was chosen for implementation, are the on-device acceleration capabilities. Typically custom code is required for delegation, however TF contains a delegate API. Usage of APIs minimizes the programming work required for delegating a model on specialized platforms. [62]

4.5.2. TensorFlow Lite

TFLite is an optimization toolkit for transforming the TF models into TFLite models, which is a compressed, smaller, faster, and efficient format. The toolkit has two main components: an interpreter and a converter. TFLite has optimized versions for most operators, but some optimizations are hardware-specific. [24]

For inference we extracted FP-32 and FX-8 models. For accuracy tests we used FP-32, FP-16 and FX-8 models. Our final model has estimated 423420 parameters.

Operations and MACs are related to input sizes. The estimated number of arithmetic operations required for inference are in Table 1.

Table 1. Estimated number of arithmetic operations

Input size	Number of arithmetic operations
QVGA [320 240]	$4.0 * 10^8$
VGA [640 480]	$1.5 * 10^9$
FHD [1920 1080]	$1.0 * 10^{10}$
DCI 4K [4096 2160]	$4.3 * 10^{10}$

4.5.3. C++

In the implementation the project was ported to a C++ environment, main reasons being ability for compilation and better portability. Compiled languages offer the ability for building executables from source files which, in terms of speed, are on a different level compared to interpreted languages. [63]

Portability in C++ means that we can build source files with one device, move the source files to another device, and the device has minimal asset requirement outside of the project. This becomes crucial when developing applications for hardware with limited computational capabilities, such as embedded and mobile devices. The compilation process is visualized in Figure 35. Preprocessor processes the macros and defined files are expanded based on declarations. The compilation process assembles the code into machine-readable formats. In the linking process, a library or an executable is produced. [63]

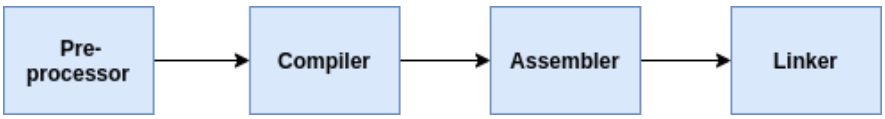

Figure 35. Compilation Process in C++.

4.5.4. Calibration

Before the evaluation phase, calibration was conducted. In this process, the model was prepared for folding by dividing operations into separate nodes. Calibration consists of two phases: calibration and model conversion.

In calibration we finalize the graph and insert fake quantization nodes. Fake quantization nodes are containers for the quantization parameters. For calibration, a small dataset was gathered. The model was finalized before quantization, meaning that the network weights are not modified. Fake quantization nodes are used for mapping quantization ranges of quantized layers. Activation ranges are saved to checkpoint files. For conversion, we used the TensorFlow Lite optimization toolkit.

5. FACE DETECTION EXPERIMENTS

In inference benchmarking there are two fundamental issues of interest. Average runtime and the initialization time. The tests concern model inference time, model size, accuracy and power consumption.

The model inference test includes two parts: optimal threading and resolution. Testing each threading option gives the optimal threading for each hardware: CPU, GPU and DSP. Tests have been carried out for each important resolution. They are

- Quarter Video Graphics Array (QVGA) [320 240],
- Video Graphics Array (VGA) [640 480],
- Full High-Definition (FHD) [1920 1080], and
- Digital Cinema Initiatives (DCI 4K) [4096 2160].

For model accuracy tests we need to select meaningful datasets. The most important attributes are consistency, scalability, and the level of difficulty. The dataset needs to contain images with multiple levels of detection challenges in various situations, so the detector required balanced data for proper benchmark scoring. Precision-recall data was used for accuracy evaluation.

Model weights and parameters of a network are saved for inference. We were interested in FP-32, FP-16 and FX-8 models, because the different hardware options support different arithmetics. During quantization information is lost and there are no distinct guidelines for model capacity and the breaking point of the model. Delegates are helper functions for accelerating models on certain platforms.

Hardware development kit 8250 (HDK8250) was used for measurements. Important characteristics for tests are computation arithmetic, delegation and hardware. Tests were conducted on:

- CPU (Qualcomm Kryo 585 CPU)

  Supports both FP and FX arithmetics. CPU calculations are operated on an OpenCL optimized solution. Qualcomm Kryo 585 consists of single 2.84GHz (Cortex A77), three 2.4GHz (Cortex A77) and four 1.8GHz (Cortex A55) cores.

- GPU (Qualcomm Adreno 650 GPU)

  Requires a FP model for inference. For GPU acceleration we use TFLite GPU delegate, which supports OpenGL.

- Mobile DSP (Qualcomm Hexagon 698 DSP)

  Requires a FX model. NNAPI and Hexagon delegates are used for accelerating on DSP. NNAPI is a general-purpose accelerator and Hexagon delegate is specific for Qualcomm devices. The DSP has a theoretical output of 15 Tera operations per second (TOPS). However, the actual processing time for an image is more meaningful metric, representing the time the model needs for inference.

The threading benchmark provides minimum-, average- and maximum inference times of the run as standard deviation.

5.1. Optimal Threading

Many TF operations support multi-threaded kernels. Like inference, threading is hardware specific. We conducted the tests for optimal number of threads on CPU inference. For GPU and DSP the optimal number of threads is 1 on all tests. Optimal thread count was found by testing at each resolution. A half-second minimum warmup time was used. The minimum number of test runs was 50. The average value of set of runs is the measurement value. Cooldown of five seconds was used between measurements. Figure 36 shows the absolute FX and FP inference durations at each test resolution.

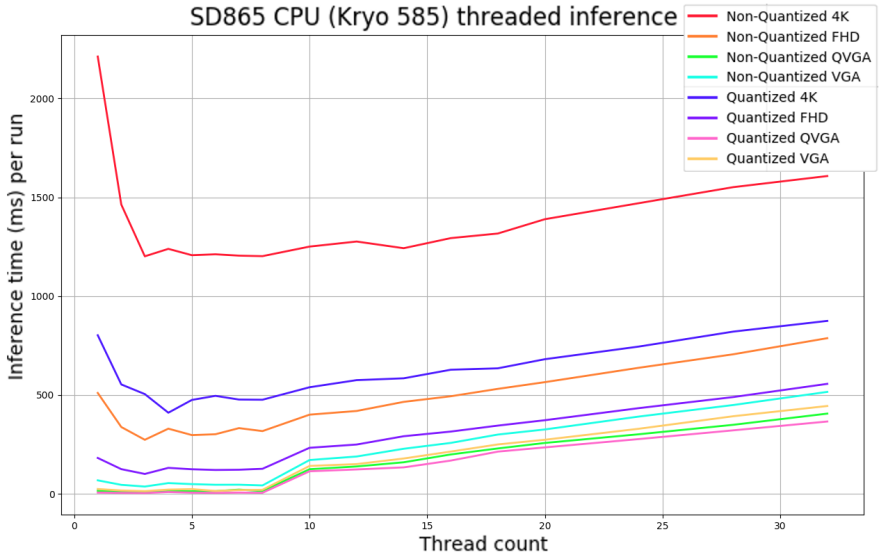

Figure 36. RetinaFace threaded inference on Qualcomm Kryo 585

For used models the larger the input image is, the more threads can be effectively utilized. However increasing the thread count increaces the amount of energy used, so there is trade-off between number of threads used and energy efficiency.

Figure 37 shows QVGA and VGA inference durations when the number of threads goes from one to eight. The fastest inference is achieved by using three threads on each resolution.

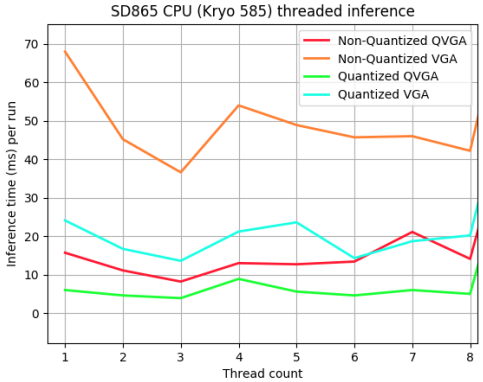

Figure 37. RetinaFace QVGA and VGA threaded inference on Qualcomm Kryo 585

In ARM big.LITTLE architecture cores are divided into two clusters: big and LITTLE. Big cores are designed for maximum throughput and LITTLE cores for maximum efficiency. LITTLE cores are utilized for minor tasks as the static leakage is smaller compared to the performance-oriented CPU design. To run task on either cluster independently, or pairing cores from clusters or with all cores visible for the task. In our tests we will use only the big cluster for maximizing stability. With larger input sizes more threads can be effectively used. $\Delta - \%$ of measurements is given by Equation (21), where x is the observed inference duration and x_r is the reference inference duration.

$$\Delta - \% = \frac{|x - x_r|}{x_r} * 100\% \tag{21}$$

The measurements were conducted on three different thread counts:

- Single thread inference,
- Balanced thread inference and

  The thread with highest delta in inference duration balances efficiency and energy usage. We observed the most energy-efficient and fastest inference cases compared to a single thread inference. The thread count was decided based on Equation (22), where x is the inference duration, x_n is the inference duration of threading, n is the number of threads and c_t is the balanced consumption test case. The plots are displayed in Appendix 2.

$$c_t = \max \frac{|x_1 - x_n|}{n} \tag{22}$$

- Maximum throughput threading.

  Maximum throughput threading was the thread count where the smallest inference duration was measured.

Threading acceleration on FP-32 model is observed in Tables 2 and 3. The FP-32 model was capable of acceleration on tested resolutions. In two-threaded case, the QVGA was slightly worse at accelerating, compared to other resolutions. In maximum throughput case, the QVGA test was the best case at acceleration.

Table 2. Balanced threading FP-32 inference acceleration

Input Size	Candidate threading	Target-thread inference	Single-thread inference	$\Delta - \%$
QVGA [320 240]	2	11.1 ms	15.7 ms	29.3 %
VGA [640 480]	2	45.2 ms	68.0 ms	33.5 %
FHD [1920 1080]	2	337.8 ms	510.2 ms	33.8 %
DCI 4K [4096 2160]	2	1463.6 ms	2212.6 ms	33.9 %

Table 3. Maximum throughput threading FP-32 inference acceleration

Input Size	Maximum throughput threading	Target-thread inference	Single-thread inference	$\Delta - \%$
QVGA [320 240]	3	8.2 ms	15.7 ms	47.8 %
VGA [640 480]	3	36.6 ms	68.0 ms	46.2 %
FHD [1920 1080]	3	273.2 ms	510.2 ms	46.5 %
DCI 4K [4096 2160]	3	1201.7 ms	2212.6 ms	45.7 %

Threading acceleration on FX-8 model is observed in Tables 4 and 5. FX acceleration was relatively better in maximum throughput 4K tests, while slightly worse in other tests. However, FX inference is, at least twice, faster in all test cases.

Table 4. Balanced threading FX-8 inference acceleration

Input Size	Candidate threading	Target-thread inference	Single-thread inference	$\Delta - \%$
QVGA [320 240]	2	4.6 ms	6.0 ms	23.3 %
VGA [640 480]	2	16.7 ms	24.1 ms	30.7 %
FHD [1920 1080]	2	124.6 ms	181.3 ms	31.3 %
DCI 4K [4096 2160]	2	553.3 ms	802.2 ms	31.0 %

Table 5. Maximum throughput threading FX-8 inference acceleration

Input Size	Maximum throughput threading	Target-thread inference	Single-thread inference	$\Delta - \%$
QVGA [320 240]	3	3.6 ms	6.0 ms	40.0 %
VGA [640 480]	3	13.6 ms	24.1 ms	43.6 %
FHD [1920 1080]	3	100.4 ms	181.3 ms	44.6 %
DCI 4K [4096 2160]	4	410.2 ms	802.2 ms	48.9 %

5.2. Accuracy Benchmarking

Loss functions expose what kind of errors are minimized in the prediction. A common metric is precision-recall (PR) that can be presented in easy to understand graphical form. Precision and recall are given by Equations (23) and (24).

$$precision = \frac{TP}{TP + FP} \tag{23}$$

$$recall = \frac{TP}{TP + FN} \tag{24}$$

Some other metrics are receiver operating characteristic (ROC), area under curve (AUC) and area under PR curve (AUCRP). PR curves are used when system is dealing with imbalanced datasets [64]. Each of these metrics has unique characteristics. PR

and ROC curves are used for performance measurements in classification problems. AUC describes model capability between classes. AUCRP is a general performance measurement for describing a complete PR curve [65]. A robust model has AUC near 1, while a model with AUC 0.5 has no capability for class separation.

Most evaluation metrics require ground truth labels. Work required for labeling a large dataset is enormous, so it is common to use public datasets for testing [66]. In Figure 38 WIDER Face dataset is used for benchmarking prediction capabilities of the selected models. Measurement goals are determination of quantization accuracy, resolution prediction capability and relevance.

Relevance is tested by comparing RetinaFace implementation against other SOTA face detector benchmarking scores. The tests measure FP-32 performance. RFB and slim [67] are lightweight single shot detection algorithms released after RetinaFace [4]. BlazeFace was tested with input sizes [256,256,3] and [128,128,3] [68].

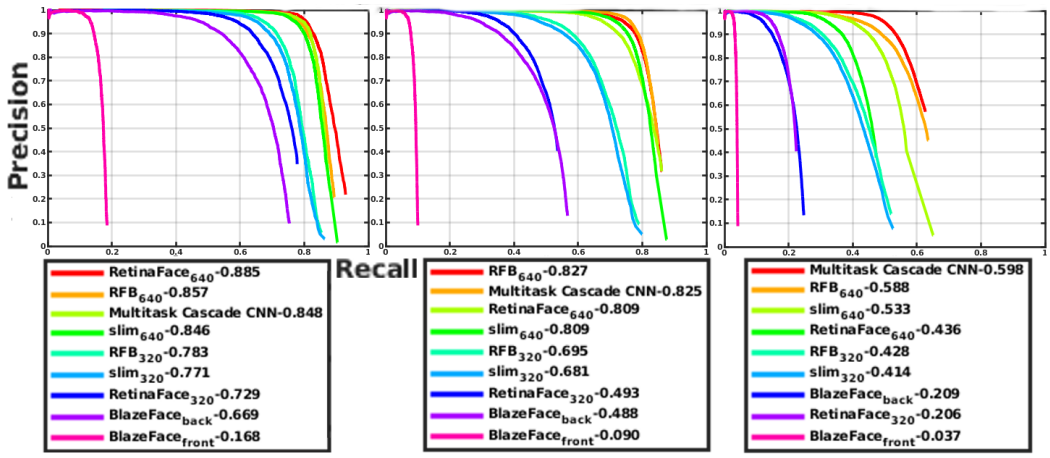

Figure 38. WIDER Face easy (left), medium (middle) and hard (right) subset precision-recall comparisons between models.

We tested the model capability resolution with following model types: FX-8, FP-16 and FP-32. The FX-8 model is post-training quantized. The resolution tests were benchmarked using WIDER Face dataset at QVGA, VGA and [1600 2150] formats. Benchmark parameters are listed in Table 6.

Table 6. Benchmark parameters used in resolution tests

Target size	Smaller dimension of image is resized to this value. For [QVGA, VGA, [1600 2150] the value is [240, 480 and 1600].
Maximum size	Defines the clipping range for larger image dimension. Small value results in changing of aspect ratio. Large value is set for keeping the aspect ratio.
Multi-scale detection	Only a single shot detection for each image is carried out.
Detection threshold	Is set to 0.02. By setting low detection threshold the detector reviews many candidate matches. Low value can result in large number of false positive detections.

FP-32 to FP-16 reduced the accuracy barely noticeably as documented in Appendix 3. However, FX-8 quantization has significant impacts on the prediction result. The accuracy drops are documented in Figure 39.

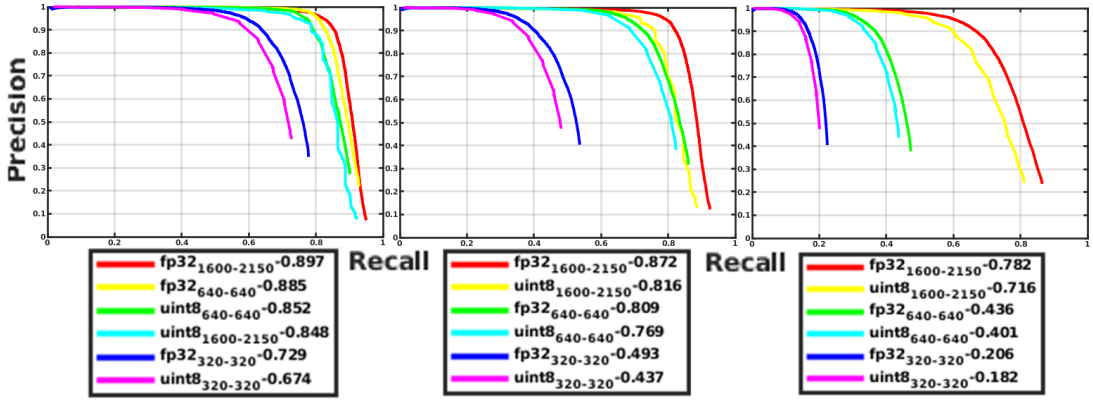

Figure 39. RetinaFace FX-8 quantization precision and recall on easy (left), medium (middle) and hard (right) subsets.

Currently trending problems with face detection deal with occlusion due to increased mask usage. This motivated a two part occlusion test: In the first test a static input size face detector is tested on VGA images, and in the second test a dynamic detector is used. Static and dynamic refer to the input image that is fed to the detector. In the face detector implementation largest stride has length of 32 pixels, so the input image dimension needs to fit into the stride in dynamic testing.

In these experiments the classifications were divided into four types:

- True positive (TP),

  A prediction is a true positive (TP) when prediction agrees, according to defined threshold value, with the ground truth value.

- False positive (FP),

  False positive (FP), or type 1 error (FP), is a prediction when there is no corresponding ground truth value present.

- False negative (FN), and

  False negative (FN), or type 2 error, occurs when condition is not detected even though the condition is presented in the ground truth value.

- True negative (TN).

  A true negative (TN) result is a result where there is no detection and there is no reason for detection.

These variables are represented in a confusion matrix. Confusion matrix summarizes the classification performance according to the test data. In a two-class problem it is 2-dimensional.

A number of evaluation metrics can be used for confusion matrices for obtaining information about prediction capability and plausible pitfalls of a prediction. Metrics for accuracy and sensitivity are defined in Equations (25) and (26).

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{25}$$

$$sensitivity = \frac{TP}{TP + FN} \tag{26}$$

The total size of the dataset was 108 images with one face per image. Database image category explanations are in Table 7.

Table 7. Benchmark categories in resolution tests

Face occlusion	light, medium and heavy occlusion. light occlusion case is a face with mask. Medium case is easy case with a hat or long frontal hair. Hard occlusion is a medium case face with mirror glasses or a mask that casts shadow over face.
Mask type	Mask types are divided into categories of common, uncommon and other. Common masks are white, light blue or black masks. Uncommon category consists of masks with patterns. Other category is invented methods for occluding face, including old diving gear, blanket, shoe and fruits.
Image Size	By image size we divide dataset into tiny (=<320), small (320<S=<480) and normal (>480) images. Category is defined by bigger image dimension in dynamic dataset face detection evaluation.
Face Size	Face size is defined as head, body and small. In head category models head is covering most of the image. Selfie or close-up still shot would be most accurate example. In body, the model would have at least chest on the picture. In tiny only a small part of the image consists of the human model.
Ambience	Ambience is controlled, normal or heavy. Example of controlled scene is photographic session background setup for faces or white background. Normal ambience is in daylight and common background such as restaurant, airport or pavement. Heavy ambience indicates the alpha of image is clearly affected by a directed point light source or ambient effect. Ambient effects are scenes with low-lightning setup. We define images by categories next.
Pose	Pose of images is always typical according to WIDER Face pose setting.
bounding box representation	We adapt face bounding box representation rules from WIDER Face evaluation scheme, where face is defined to tightly contain forehead and chin. [3]

Dataset is numerically categorized in Table 8. The data contains multiple difficulty levels, various scenarios, and occlusion types. The dataset is created for having a reliable, but compact, validation set.

Table 8. Occlusion dataset images categorized

Face Occlusion	54 (Light)	40 (Medium)	14 (Heavy)
Mask Type	67 (Common)	28 (Uncommon)	13 (Other)
Image Size	88 (Tiny)	6 (Small)	14 (Normal)
Face Size	66 (Head)	34 (Body)	8 (Small)
Ambience	44 (Controlled)	58 (Normal)	6 (Heavy)

Dynamic FX-8 model test result is in Figure 40, where p' and n' are the predicted values, and p and n are the ground-truth values. In dynamic detection occlusion test the most important notations are sensitivity and false negative rate (FNR). Sensitivity is 88.89% for our predictions and FNR is 11.11%. However, the input image and face sizes have significant impacts on the prediction robustness.

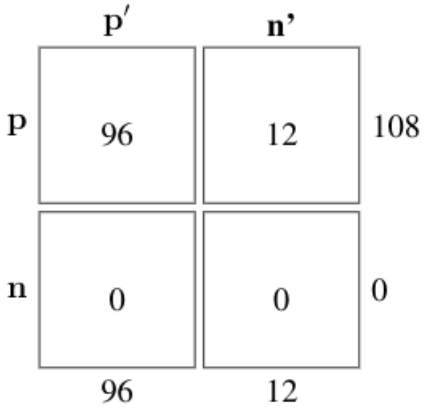

Figure 40. Occlusion test confusion matrix with dynamic input size.

Static FX-8 VGA mode test result is in Figure 41. Sensitivity of static occlusion dataset run is 96.26% and FNR only 3.74%, while the precision is 95.37%. After resizing the input size to VGA, the sensitivity and FNR of detections are better than detection scores at original image sizes. However, due to the change of input size, a new problem has emerged. Because of the restrictions of the NMS, some detections are only coarsely at the ground-truth locations, resulting in type 1 errors.

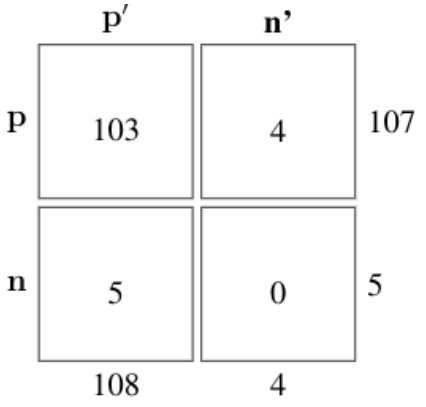

Figure 41. Occlusion test confusion matrix with static input size.

5.3. Detection Ranges

When we make a prediction on an image, the network indicates face representations. A relevant question is what are the smallest and largest detectable faces, and how well the detector performs in non-ideal situations, where the face is noisy, pixelated or deformed. The fundamental variables are absolute and relative sizes of the face and the image, which are presented in Figure 42.

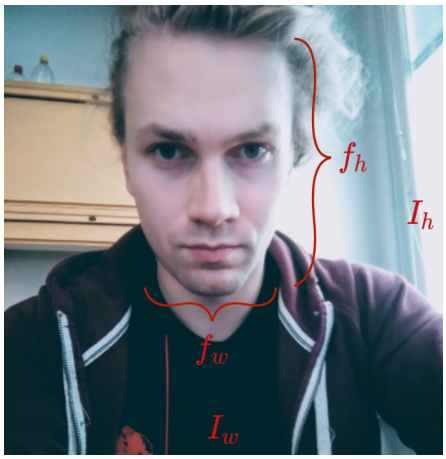

Figure 42. The variables in the face tests.

The next phase in testing was changing the image-to-face ratio. The image-to-face ratio was scaled stepwise by factor of 0.5 until a detection failed. We are also interested on the robustness of the detector in case of differences between the FP-32 and the FX-8 model predictions at different scales. Figure 43 is an example of a test suite for determining minimum face scale detection range.

Figure 43. An example of minimum prediction test.

The smallest detection was 25-by-30 pixels with FP-32 model. However, with lower quality input data the results deteriorated.

For better representation of detection capability the worlds largest selfie was tested. The prediction threshold was set to 0.1. Red indicates confident detection and the more yellow the bounding box becomes, the less confident the prediction is. The detections

were predicted on the image on original input size, which is 2048-by-1150 pixels. The FP-32 model detections are in Figure 44. The total number of faces in the image was 1151. The total number of detections is 737, but there are false positives among the predictions.

Figure 44. FP-32 model detections on the World's Largest Selfie powered by Lumia 730.

With the FX-8 model, the face detector lost small sized faces. Quantized detections are observed in Figure 45. The total number of detections was 456.

Figure 45. FX-8 model detections on the World's Largest Selfie powered by Lumia 730.

5.4. Inference Benchmarking

Measurements consisted of 250 inferences, for the tests TFLite Android benchmarking tool was used. In every measurement maximum throughput threading was used. CPU inference measurements are in Table 9.

Table 9. CPU inference benchmarking

Input Size	FX-8	FP-32	Δ-%
QVGA [320 240]	3.5 ms	8.0 ms	56.3 %
VGA [640 480]	13.4 ms	36.7 ms	63.5 %
FHD [1920 1080]	99.1 ms	270.9 ms	63.4 %
DCI 4K [4096 2160]	395.7 ms	1051.1 ms	62.4 %

GPU inference benchmark is visualized in Table 10. In TFLite inference benchmarking FP-16 inference is tested by using all GPU FP cores. Typically mid-range smartphone GPUs do not gain acceleration by usage of all cores.

Table 10. GPU inference benchmarking

Input Size	FP-16	FP-32	Δ-%
QVGA [320 240]	11.2 ms	12.1 ms	7.4 %
VGA [640 480]	18.1 ms	28.1 ms	35.6 %
FHD [1920 1080]	79.8 ms	122.1 ms	34.6 %
DCI 4K [4096 2160]	244.1 ms	407.1 ms	40.0 %

For DSPs there is an option to use FX-8 model on full and fallback quantization. In fallback mode unsupported operations are executed by the CPU. Results are observed in Table 11.

Table 11. DSP inference benchmarking

Input Size	Full FX-8	Fallback FX-8	Δ-%
QVGA [320 240]	3.4 ms	5.6 ms	39.3 %
VGA [640 480]	16.5 ms	28.6 ms	42.3 %
FHD [1920 1080]	65.0 ms	136.8 ms	52.5 %
DCI 4K [4096 2160]	314.7 ms	569.4 ms	44.7 %

The initialization times are documented in Table 12. The initialization time for the model starts when the system begins to load the input image and ends when the processing of inference begins. For initialization times averages at all resolutions are measured. CPU and DSP measurements are on FX-8 models and GPU measurements on FP models. Initialization times are the means of 10 runs.

Table 12. Initialization times

Input Size	CPU FX-8	GPU FP-32	GPU FP-16	DSP FX-8
QVGA [320 240]	1.1 ms	661.9 ms	826.0 ms	107.5 ms
VGA [640 480]	0.4 ms	681.2 ms	826.0 ms	103.7 ms
FHD [1920 1080]	0.5 ms	727.4 ms	857.5 ms	203.1 ms
DCI 4K [4096 2160]	0.4 ms	924.2 ms	940.0 ms	481.2 ms

Additional benchmarks for mid-range chipsets are in Appendix 4.

6. POWER DISSIPATION ANALYSIS

In this chapter energy efficiency of the face detection algorithm on CPU, GPU, and DSP is analyzed. The test cases determine energy usage to find a trade-off against the inference and performance. Factors inducing noise and bias to the measurements are presented in Appendix 5.

6.1. Measurement Environment Setup

Chipset SnapDragon 865 (SD865) was used for the measurements and big.LITTLE architecture was optimized for stability. FX-8 models were tested with the CPU and DSP, and FP-32 and FP-16 models were tested with the GPU.

The energy dissipation is calculated as means over a period of time. In practice, minute-long inference sessions were run for gathering the data points at 898.08 Hz. Power consumptions are measured using a National Instruments USB-4065 device and are performed using TFLite Android benchmarking application. For visualization purposes, a Gaussian filter and smoothing have been applied to the data. The raw data visualization is in Appendix 6.

6.1.1. CPU Measurement Visualization

CPU energy dissipation was measured for single-threaded (T1), balanced per-thread (TB) and maximized throughput threading (TT) cases based on Tables 4 and 5. The fastest threading consumes significantly more energy per duration compared to other measurements, but the duration is shorter. The measurements are shown in Figure 46.

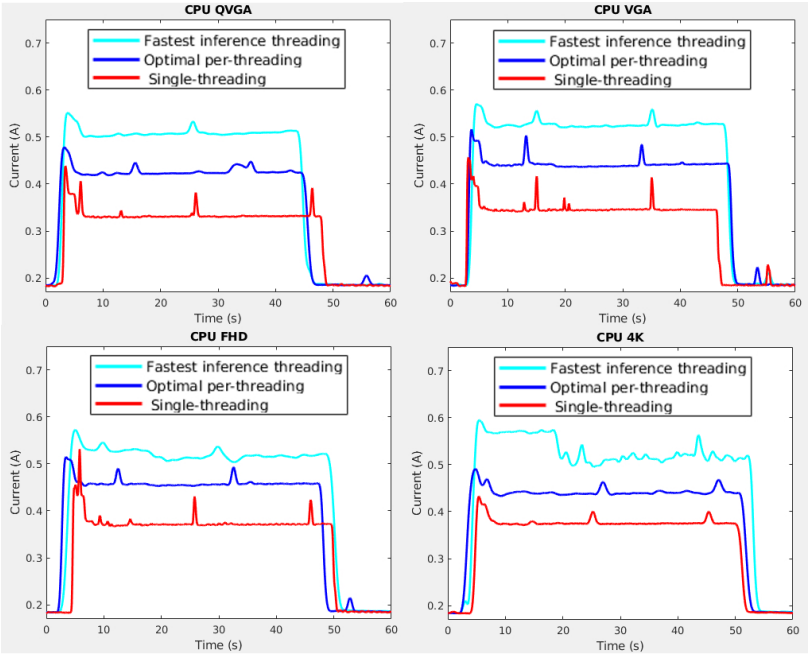

Figure 46. CPU FX-8 power dissipation visualization.

6.1.2. GPU and DSP Delegation Visualization

Figure 47 contains GPU tests for the FP-32 and FP-16 models, and DSP test for the FX-8 model. The initialization part is the peak of energy consumption, which limits the number of inferences we were able to run for each test benchmark.

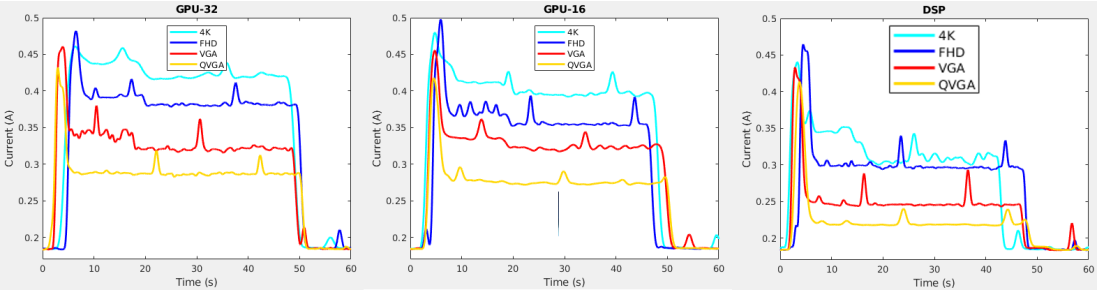

Figure 47. GPU FP-32 and FP-16, and DSP FX-8 power dissipation visualization.

6.2. Quantitative Power Dissipation Analysis

Evaluation of measurements visualized in Figures 46 and 47 takes place in this section. The goals of the analysis are the power consumption of battery use case and per inference energy dissipation. The measurements are for model inferences only.

6.2.1. Initial Setup for Measurements

Per-inference duration is the total time divided by the count of inferences. Setup for all variables needed for measurements are in Appendix 7. (Recall Tables 10 and 11.) In Table 13 the actual per inference durations are displayed. The initialization latency impacted the number of inference runs in GPU and DSP tests.

Table 13. Actual Per-inference duration

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	6.1 ms	23.9 ms	184.5 ms	833.5 ms
CPU FX-8 TB	4.3 ms	16.8 ms	130.3 ms	612.5 ms
CPU FX-8 TT	**3.8 ms**	**14.0 ms**	102.7 ms	445.7 ms
GPU FP-32	13.9 ms	30.1 ms	143.5 ms	494.6 ms
GPU FP-16	11.5 ms	19.4 ms	98.6 ms	**314.3** ms
DSP FX-8	12.5 ms	19.8 ms	**81.0 ms**	322.5 ms

6.2.2. Energy Consumption Tests

Wattage of the system is calculated in Table 14.

Table 14. Per-inference energy consumption

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	1.8 W	1.9 W	2.3 W	2.3 W
CPU FX-8 TB	2.9 W	3.1 W	3.3 W	3.1 W
CPU FX-8 TT	3.9 W	4.1 W	4.0 W	4.1 W
GPU-32	1.2 W	1.6 W	2.2 W	2.6 W
GPU-16	1.0 W	1.6 W	1.9 W	2.4 W
DSP FX-8	**0.5 W**	**0.8 W**	**1.3 W**	**1.5 W**

Usually, 40 or more frames per second (FPS), or less than 25 ms per inference, is the requirement for a real-time operation. However, the device has to do other tasks in parallel, and also the dissipation needs to be taken into account.

In most applications, we don't need the maximum rate inference. Detections can be made in the beginning and then the face can be tracked. Many tracking algorithms are cheaper to calculate compared to real-time detection. This saves resources and limits the representation space.

Energy-inference trade-off was concluded in a real-time detection case and a theoretical calculation on battery consumption in-device. Most of the time energy consumption in still-image detection is not as vital as in real-time video detection.

6.2.3. Estimated Power Consumption on Smartphones

Typical lithium-ion battery provides 2.8-4.2 V. Temperature, age, optimization techniques, and discharge capacity affect the voltage. 12 V direct current (DC) was converted to a nominal charge of 3.7 V DC. It is preferred to draw as little current as possible from the battery to maximize its capacity. The results are in Table 15.

Table 15. Estimated Electrical Current on 3.7 V DC

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	0.4836 A	0.5234 A	0.6142 A	0.6214 A
CPU FX-8 TB	0.7748 A	0.8309 A	0.8821 A	0.8274 A
CPU FX-8 TT	1.042 A	1.1017 A	1.079 A	1.1079 A
GPU-32	0.3133 A	0.4301 A	0.5863 A	0.7008 A
GPU-16	0.2805 A	0.4294 A	0.5235 A	0.6418 A
DSP FX-8	**0.1249 A**	**0.2053 A**	**0.3490 A**	**0.4164 A**

Per-inference energy consumption is calculated from the mean consumption measurements, voltage and per-inference times. The outcome is in Table 16.

Table 16. Estimated Power Dissipation per Inference

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	10.9 mJ	46.3 mJ	419.3 mJ	1916.4 mJ
CPU FX-8 TB	12.3 mJ	51.6 mJ	425.3 mJ	1875.1 mJ
CPU FX-8 TT	14.7 mJ	57.1 mJ	410.0 mJ	1827.0 mJ
GPU-32	16.1 mJ	47.9 mJ	311.3 mJ	1282.5 mJ
GPU-16	11.9 mJ	30.8 mJ	191.0 mJ	746.4 mJ
DSP FX-8	**5.8 mJ**	**15.0 mJ**	**104.6 mJ**	**496.9 mJ**

7. DISCUSSION

In this thesis, a functional face detector was ported to CPU, GPU, and DSP. The evaluated model was optimized to the ARM platform. The implementation was not thoroughly optimized and the results in this thesis represent work in progress.

7.1. Initialization Pipelining

Initialization optimization requires knowledge of programming languages that are used for shaders and kernel computation on runtime. Typically the open computing language (OpenCL) and open graphics language (OpenGL) libraries are supported by most smartphones, OpenGL being responsible for moving shaders to binary format and OpenCL for creating kernels.

Initialization optimization is context-dependent. We need to have a clear view for figuring an optimal way as the requirements vary for still-image initialization and video tracking. In video tracking the optimal case could be to not detect until other processes have finished, resulting in minimal latency on the system. In the still-image case, the best approach is likely to have high priority on the inference so the user can observe detection as soon as possible.

Initialization times on DSP and GPU can be substantially lowered if input-output memory management unit (IOMMU) memory allocation can be used for mapping data straight to DSP or GPU memory space. This way additional initialization is not required as large regions of memory can be allocated to devices without massive overheads in buffer copying. Another possibility would be mapping data to shared memory, but this is not a widely used method.

7.2. About Evaluation

Base metrics used in the evaluation phase are mean and standard deviation on a set of inferences. The calculated mean is most often relatively close to the minimum inference value in inference benchmarks, so the maximum values are outliers, which increases mean inference duration substantially. It would be more accurate to use the minimum time for inference. This way we can evaluate the system based on hardware capability and based on the current experience the minimum values between runs are reduced in variance compared to mean values statistics.

7.3. New Detection Technologies

The phase in which new technologies and techniques emerge in artificial intelligence is phenomenal. New detectors are being developed and rapidly published around the globe. It is already observable that this implementation is not the fastest and most reliable detector in lightweight detection in terms of some of the benchmarks but the results are overall in a good balance for a detector, especially for VGA images.

8. CONCLUSION

The aim of this thesis was to train, develop, and test a state-of-the-art face detection algorithm and to create a balanced version in terms of latency, energy efficiency, and accuracy. Optimal inference-accuracy trade-off was achieved by using a MobileNet-based single shot face detector. The original project was developed and trained in the MXNet [69] framework. Later the network was converted to TensorFlow framework and deployed using the TensorFlow Lite toolkit. Compactibility of a network has become important part of design on embedded and mobile platforms, especially for applications requiring real-time tracking.

The most coherent inference acceleration method was fixed-point quantization. Inference was further accelerated by fine-grained multithreading. Multithreading accelerated the inference, but only on a couple of threads. Based on the evaluation, latency improvements added to energy efficiency and the faster the inference duration, the smaller the energy cost. An important aspect of energy efficiency improvements was model delegation. Delegation to DSPs and GPUs minimizes energy consumption compared to CPUs, but initialization durations were challenging. Best energy efficiency was achieved by DSP delegation. Input size affects latency, inference duration and initialization durations. With tested input sizes the computation cost grew quadratically, so minimization of the input size required for a precise detection was important.

Floating point model conversion to a smaller bit-depth provided minimal losses in accuracy. Fixed-point quantization was conducted post-training and provided lossy, but capable results after quantization to 8-bit representation. Precision and recall of trained models were tested using the Wider Face dataset. Testing included multiple image sizes and all results displayed prominent results, but the smaller the image size, the detections deteriorated. Trained model capability with VGA input size performed well for easy category, but for medium and hard categories we were able to find a better performing detector. For QVGA input size RFB and slim detectors achieved overall better precision-recall scores.

Occluded face detection capability of quantized model was tested on a compact dataset, and the results were promising with small image sizes. The challenges for a dynamic and static detection were different. Detection with dynamic input size provided higher FNR rate, and detection tests conducted on static VGA input size provided higher FPR. Sensitivity of the static detection tests provided overall better results.

9. REFERENCES

[1] Huang T. (1996) Computer vision: Evolution and promise.

[2] Zou Z., Shi Z., Guo Y. & Ye J. (2019) Object detection in 20 years: A survey. arXiv preprint arXiv:1905.05055.

[3] Yang S., Luo P., Loy C.C. & Tang X. (2016) Wider face: A face detection benchmark. In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 5525–5533.

[4] Deng J., Guo J., Zhou Y., Yu J., Kotsia I. & Zafeiriou S. (2019) Retinaface: Single-stage dense face localisation in the wild. arXiv preprint arXiv:1905.00641.

[5] Harris C.G., Stephens M. et al. (1988) A combined corner and edge detector. In Alvey vision conference, vol. 15, Citeseer, vol. 15, pp. 10–5244.

[6] Canny J. (1986) A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence , pp. 679–698.

[7] Lowe D.G. (1999) Object recognition from local scale-invariant features. In Proceedings of the seventh IEEE international conference on computer vision, vol. 2, Ieee, vol. 2, pp. 1150–1157.

[8] Bay H., Tuytelaars T. & Van Gool L. (2006) Surf: Speeded up robust features. In European conference on computer vision, Springer, pp. 404–417.

[9] Dalal N. & Triggs B. (2005) Histograms of oriented gradients for human detection. In 2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05), vol. 1, IEEE, vol. 1, pp. 886–893.

[10] Liu L., Ouyang W., Wang X., Fieguth P., Chen J., Liu X. & Pietikäinen M. (2020) Deep learning for generic object detection: A survey. International journal of computer vision 128, pp. 261–318.

[11] Felzenszwalb P., McAllester D. & Ramanan D. (2008) A discriminatively trained, multiscale, deformable part model. In 2008 IEEE conference on computer vision and pattern recognition, IEEE, pp. 1–8.

[12] Papageorgiou C. & Poggio T. (2000) A trainable system for object detection. International journal of computer vision 38, pp. 15–33.

[13] Viola P., Jones M. et al. (2001) Robust real-time object detection. International journal of computer vision 4, p. 4.

[14] Viola P. & Jones M. (2001) Rapid object detection using a boosted cascade of simple features. In Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001, vol. 1, IEEE, vol. 1, pp. I–I.

[15] Ahonen T., Hadid A. & Pietikäinen M. (2004) Face recognition with local binary patterns. In European conference on computer vision, Springer, pp. 469–481.

[16] Chang-Yeon J. (2008) Face detection using lbp features. Final Project Report 77, pp. 1–4.

[17] Goodfellow I., Bengio Y., Courville A. & Bengio Y. (2016) Deep learning, vol. 1. MIT press Cambridge.

[18] Koehrsen W. (2018) Overfitting vs. underfitting: A complete example. Towards Data Science.

[19] Nilsson N.J. (1997) Introduction to machine learning.

[20] Haykin S.S. et al. (2009), Neural networks and learning machines/Simon Haykin.

[21] Agarap A.F. (2018), Deep learning using rectified linear units (relu). arXiv preprint arXiv:1803.08375.

[22] Ramachandran P., Zoph B. & Le Q.V. (2017) Searching for activation functions. arXiv preprint arXiv:1710.05941.

[23] Ioffe S. & Szegedy C. (2015) Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

[24] Jacob B., Kligys S., Chen B., Zhu M., Tang M., Howard A., Adam H. & Kalenichenko D. (2018) Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2704–2713.

[25] Krishnamoorthi R. (2018) Quantizing deep convolutional networks for efficient inference: A whitepaper. arXiv preprint arXiv:1806.08342.

[26] Smith S.W. et al. (1997) The scientist and engineer's guide to digital signal processing.

[27] TensorFlow (2020), Tensorflow model optimization guide overview. URL: `https://www.tensorflow.org/model_optimization`. Accessed 28.10.2020.

[28] Distiller (2020), Quantization algorithms. URL: `https://nervanasystems.github.io/distiller/algo_quantization.html`. Accessed 30.9.2020.

[29] Jacob B. et al. (2017), gemmlowp: a small self-contained low-precision gemm library.(2017).

[30] Sheng T., Feng C., Zhuo S., Zhang X., Shen L. & Aleksic M. (2018) A quantization-friendly separable convolution for mobilenets. In 2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2), IEEE, pp. 14–18.

[31] Zhu M. & Gupta S. (2017) To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv preprint arXiv:1710.01878.

[32] Mao H., Han S., Pool J., Li W., Liu X., Wang Y. & Dally W.J. (2017) Exploring the regularity of sparse structure in convolutional neural networks arXiv preprint arXiv:1705.08922.

[33] Han S., Pool J., Tran J. & Dally W. (2015) Learning both weights and connections for efficient neural network. In Advances in neural information processing systems, pp. 1135–1143.

[34] TensorFlow (2020), Pruning comprehensive guide. URL: `https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide`. Accessed 7.12.2020.

[35] Faraoun K. & Boukelif A. (2006) Neural networks learning improvement using the k-means clustering algorithm to detect network intrusions. INFOCOMP Journal of Computer Science 5, pp. 28–36.

[36] TensorFlow (2020), Tensorflow core tensor guide. `https://www.tensorflow.org/guide/tensor`. Accessed 28.10.2020.

[37] Stokes J. (2007) Inside the machine: an illustrated introduction to microprocessors and computer architecture. No Starch Press.

[38] Oberstar E.L. (2007) Fixed-point representation & fractional math. Oberstar Consulting 9.

[39] Scott T.J. (1991) Mathematics and computer science at odds over real numbers. ACM SIGCSE Bulletin 23, pp. 130–139.

[40] Muller J.M., Brisebarre N., De Dinechin F., Jeannerod C.P., Lefevre V., Melquiond G., Revol N., Stehlé D., Torres S. et al. (2018) Handbook of floating-point arithmetic, vol. 1. Springer.

[41] Kalamkar D., Mudigere D., Mellempudi N., Das D., Banerjee K., Avancha S., Vooturi D.T., Jammalamadaka N., Huang J., Yuen H. et al. (2019) A study of bfloat16 for deep learning training. arXiv preprint arXiv:1905.12322.

[42] Constantinides G.A., Cheung P.Y. & Luk W. (2003) Synthesis of saturation arithmetic architectures. ACM Transactions on Design Automation of Electronic Systems (TODAES) 8, pp. 334–354.

[43] Ben-Nun T. & Hoefler T. (2019) Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. ACM Computing Surveys (CSUR) 52, pp. 1–43.

[44] Amdahl G.M. (1967) Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, pp. 483–485.

[45] Huisman J.A. (2010) High-speed parallel processing on cuda-enabled graphics processing units. Ph.D. thesis, Citeseer.

[46] Chou C.C., Deng Y.F., Li G. & Wang Y. (1995) Parallelizing strassen's method for matrix multiplication on distributed-memory mimd architectures. Computers & Mathematics with Applications 30, pp. 49–69.

[47] Szegedy C., Liu W., Jia Y., Sermanet P., Reed S., Anguelov D., Erhan D., Vanhoucke V. & Rabinovich A. (2015) Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 1–9.

[48] Li C., Yang Y., Feng M., Chakradhar S. & Zhou H. (2016) Optimizing memory efficiency for deep convolutional neural networks on gpus. In SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, pp. 633–644.

[49] Golub G. & Van Loan C. (2013) Matrix Computations 4th Edition The Johns Hopkins University Press. Baltimore, MD.

[50] Wei X., Liang Y., Li X., Yu C.H., Zhang P. & Cong J. (2018) Tgpa: tile-grained pipeline architecture for low latency cnn inference. In Proceedings of the International Conference on Computer-Aided Design, pp. 1–8.

[51] Hill P., Zamirai B., Lu S., Chao Y.W., Laurenzano M., Samadi M., Papaefthymiou M., Mahlke S., Wenisch T., Deng J. et al. (2018) Rethinking numerical representations for deep neural networks arXiv preprint arXiv:1808.02513.

[52] Flynn M.J. (1972) Some computer organizations and their effectiveness. IEEE transactions on computers 100, pp. 948–960.

[53] Flynn M.J. (1966) Very high-speed computing systems. Proceedings of the IEEE 54, pp. 1901–1909.

[54] Mutlu O. (2020), Computer architecture: Simd and gpus (part iii). URL: https://course.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.1.3-simd-and-gpus-part3-vliw-dae-systolic.pdf. Accessed 27.11.2020.

[55] Zhang J., Yeung S.H., Shu Y., He B. & Wang W. (2019) Efficient memory management for gpu-based deep learning systems. arXiv preprint arXiv:1903.06631.

[56] Skolnick D. (1997) Dsp 101 part 1: An introductory course in dsp system design. Analog Dialogue.

[57] Tselepis I., Bekakos M., Milovanovic I. & Milovanovic E. (2007) Fpga implementation of optimal planar systolic arrays for orthogonal matrix multiplication. In Proceedings of the 4th International Conference: Sciences of Electronics, Technologies of Information and Telecommunications (SETIT).(Hammamet, Tunisia).

[58] Olsen E.B. (2017) Proposal for a high precision tensor processing unit. arXiv preprint arXiv:1706.03251.

[59] Abadi M., Barham P., Chen J., Chen Z., Davis A., Dean J., Devin M., Ghemawat S., Irving G., Isard M. et al. (2016) Tensorflow: A system for large-scale machine learning. In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), pp. 265–283.

[60] Rezatofighi H., Tsoi N., Gwak J., Sadeghian A., Reid I. & Savarese S. (2019) Generalized intersection over union: A metric and a loss for bounding box regression. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 658–666.

[61] Neubeck A. & Van Gool L. (2006) Efficient non-maximum suppression. In 18th International Conference on Pattern Recognition (ICPR'06), vol. 3, IEEE, vol. 3, pp. 850–855.

[62] TensorFlow (2020), Tensorflow lite performance guide. `https://www.tensorflow.org/lite/performance/delegates`. Accessed 4.11.2020.

[63] Stroustrup B. (2000) The C++ programming language. Pearson Education India.

[64] Kendrick Boyd K.H.E. & Page C.D. (2013) Area under the Precision-Recall Curve: Point Estimates and Confidence Intervals. Springer. ECML PKDD Part 3, LNAI 8190, pp. 451-466.

[65] Narkhede S. (2018) Understanding AUC-ROC Curve. Towards Data Science, 26.

[66] Jain V. & Learned-Miller E. (2010) Fddb: A benchmark for face detection in unconstrained settings. Tech. Rep. UM-CS-2010-009, University of Massachusetts, Amherst.

[67] Linzaer (2020), Ultra-light-fast-generic-face-detector-1mb. URL: `https://github.com/Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB`. Accessed 7.12.2020.

[68] Bazarevsky V., Kartynnik Y., Vakunov A., Raveendran K. & Grundmann M. (2019) Blazeface: Sub-millisecond neural face detection on mobile gpus. arXiv preprint arXiv:1907.05047.

[69] Chen T., Li M., Li Y., Lin M., Wang N., Wang M., Xiao T., Xu B., Zhang C. & Zhang Z. (2015) Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274.

10. APPENDICES

Appendix 1	Folding and quantization process visualization.
Appendix 2	SD865 threading visualizations.
Appendix 3	FP-32 to FP-16 quantization results on WIDER Face.
Appendix 4	Additional mid-range threading and DSP benchmarks.
Appendix 5	Biased measurement examples on power dissipation analysis.
Appendix 6	Raw power dissipation measurements on SD865.
Appendix 7	Power dissipation measurement setup.

Folding and quantization steps are represented in Figures 1, 2 and 3, where both weights and biases are folded. The visualizations are taken from Netron, which is a neural network visualization tool.

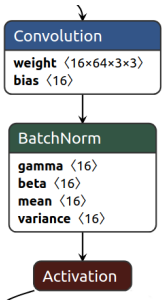

Figure 1. Initial layer setup: operations are on separate layers.

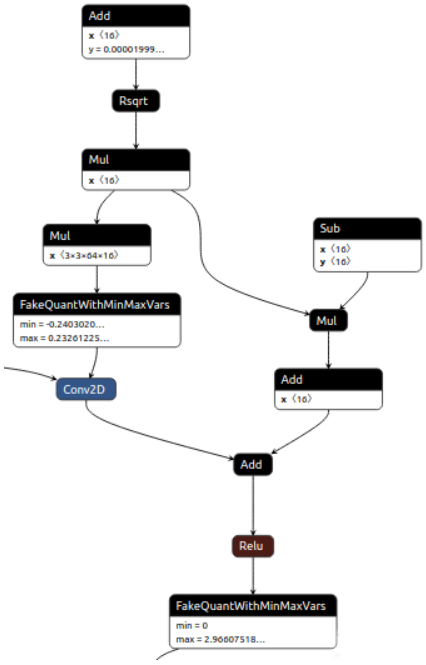

Figure 2. Quantization of a block illustrated as a graph representation.

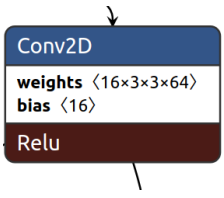

Figure 3. Folding of layers: the result is a single layer.

The balanced threading inference durations on SnapDragon 865 (SD865) CPU are visualized in Figure 1.

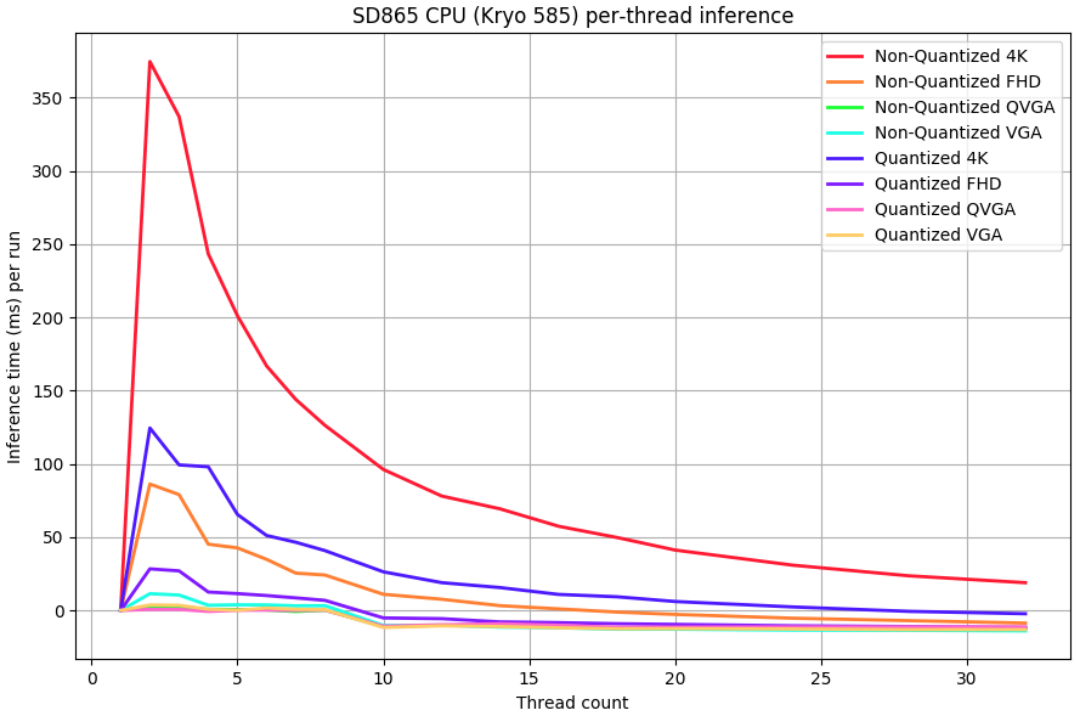

Figure 1. SD865 CPU per-thread inference.

Small input sizes are observed in Figure 2.

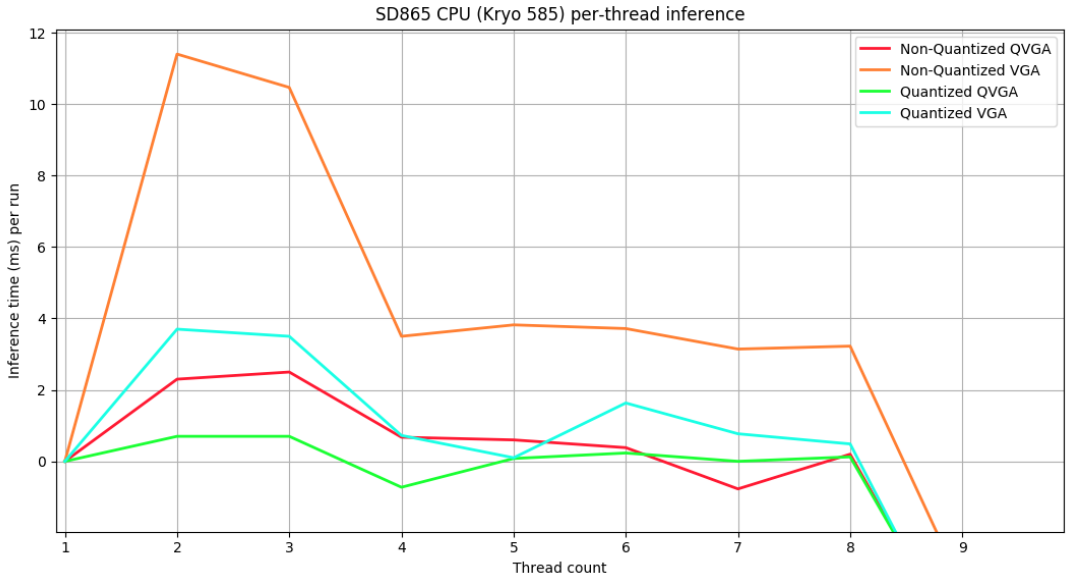

Figure 2. SD865 CPU threaded inference

WIDER Face FP-32 to FP-16 quantization results are plotted as precision-recall curves in Figures 1, 2 and 3. The FP-32 to FP-16 quantization has minimal effect on the classification precision and recall.

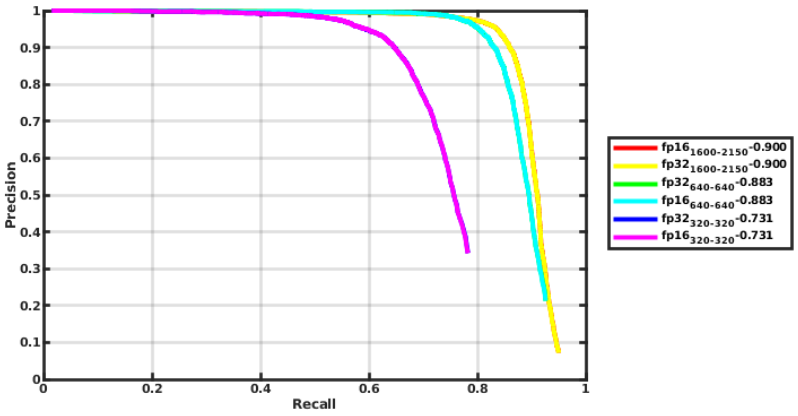

Figure 1. WIDER Face RetinaFace FP quantization on easy subset

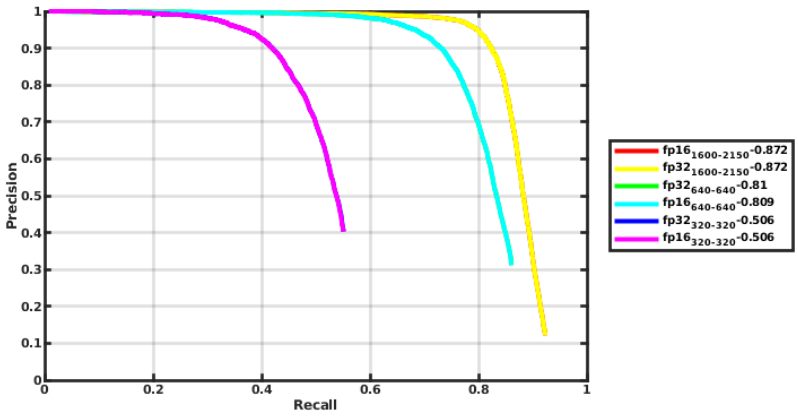

Figure 2. WIDER Face RetinaFace FP quantization on medium subset

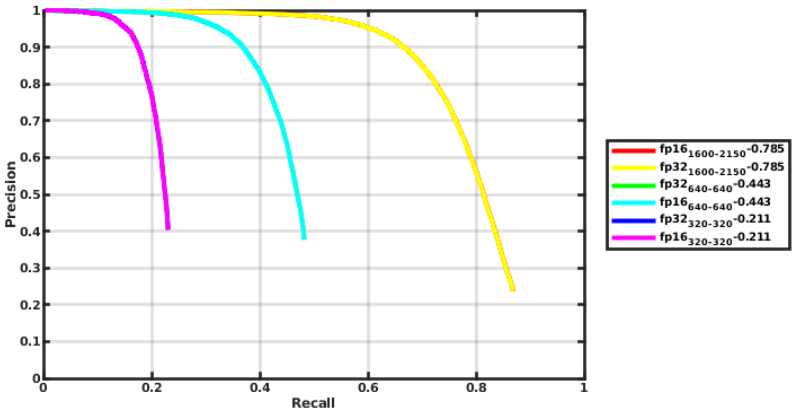

Figure 3. WIDER Face RetinaFace FP quantization on hard subset

SD865 is a high-end chipset. Additional tests are concluded on mid-range chipsets MediaTek 6883 (MT6883) and SnapDragon 765 (SD765).

The MT6883 chipset contains an octa-core CPU which consists of four ARM Cortex-A76s and four ARM Cortex-A55s, which all run on 2.0GHz. The absolute inference times on MT6883 are observed in Figure 1.

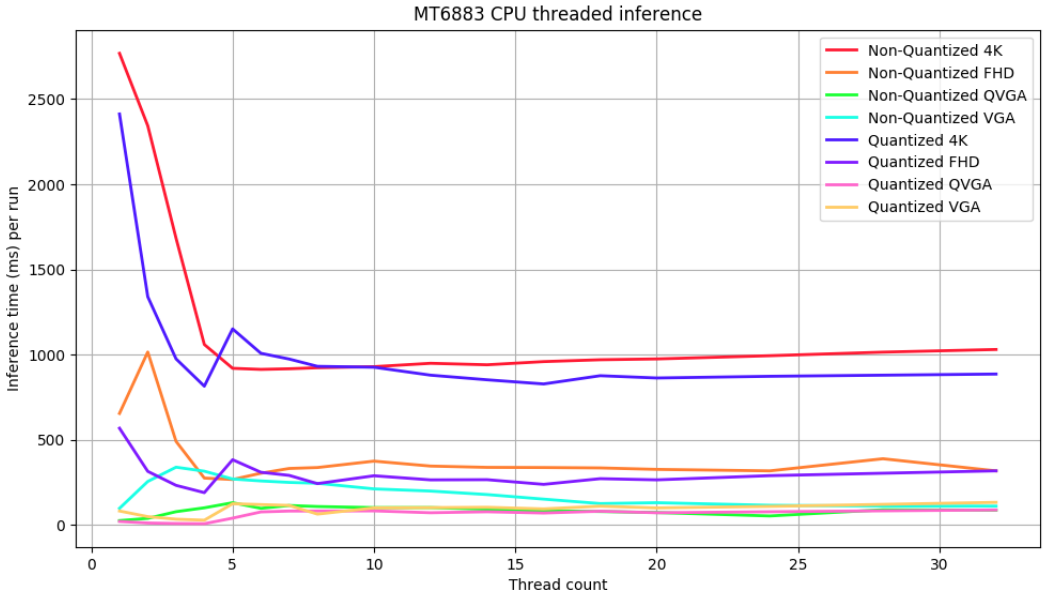

Figure 1. MT6883 CPU absolute inference times

The balanced threaded inference times are plotted in Figure 2.

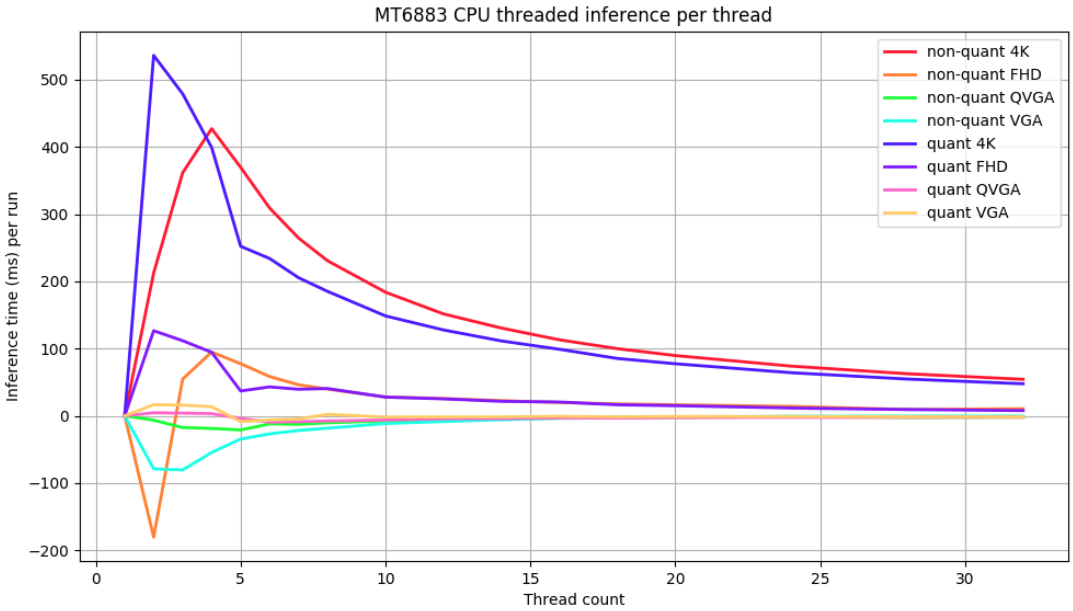

Figure 2. MT6883 CPU per thread inference times

SD765 contains a Kryo 475 octa-core CPU, which has two ARM Cortex-A76s and six ARM Cortex-A55s. Former Cortex-A76 operates on up to 2.4 GHz rate and the latter on up to 2.2 GHz rate. A-55s operate at 1.8 GHz rate. The absolute inference times on SD765 are in Figure 1.

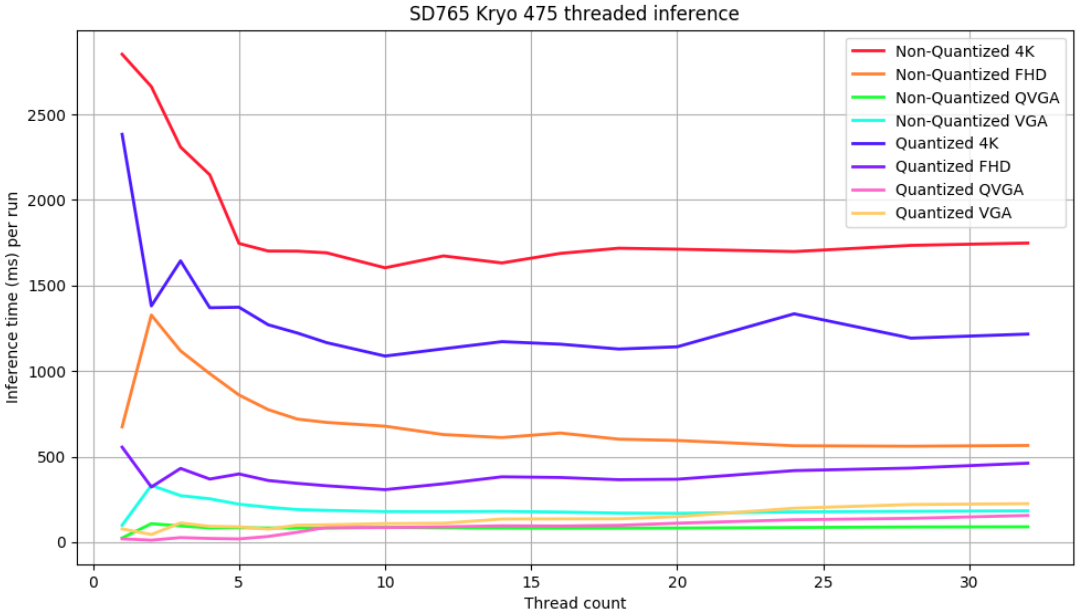

Figure 3. SD765 CPU absolute inference times

The balanced threaded inference times are visualized in Figure 2.

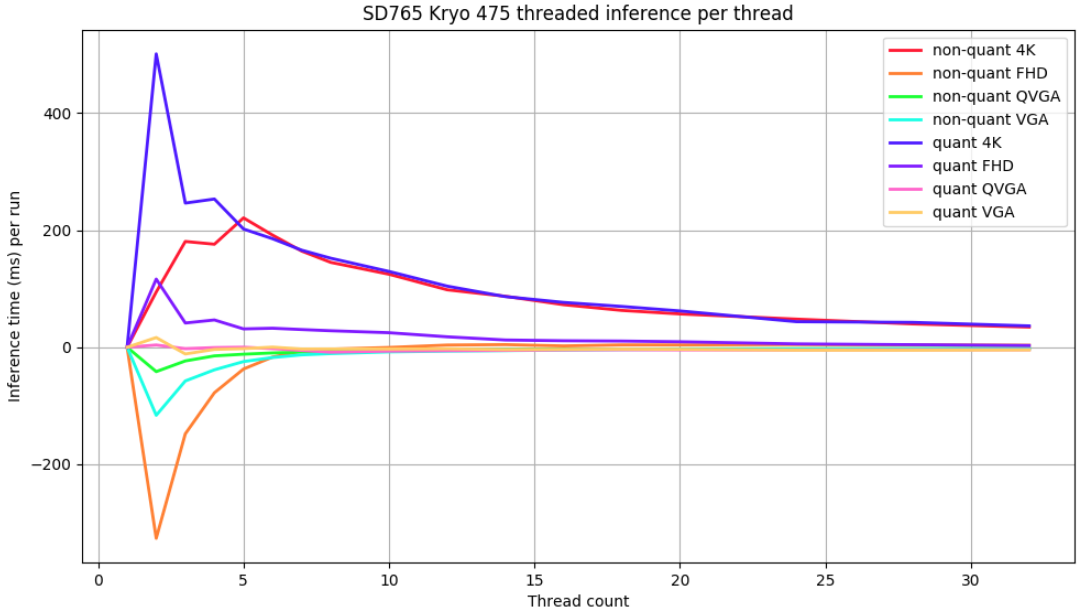

Figure 4. SD765 CPU per thread inference times

The DSP on MT6883 is not capable of inference acceleration by fine-grained threading on the TFLite Benchmarking tool with NNAPI delegate. MT6883 DSP threading is observed in Figure 1.

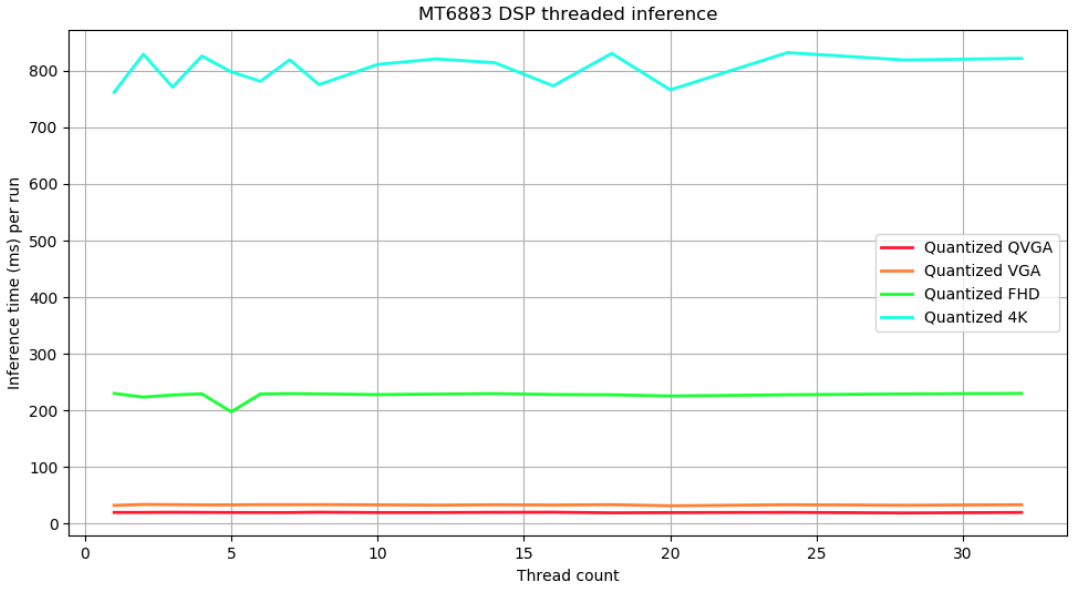

Figure 1. MT6883 DSP optimal threading

Fine-grained threading applied on the Hexagon 696 does not accelerate computation in the TFLite Benchmarking tool with NNAPI delegation. The DSP test is visualized in Figure 2.

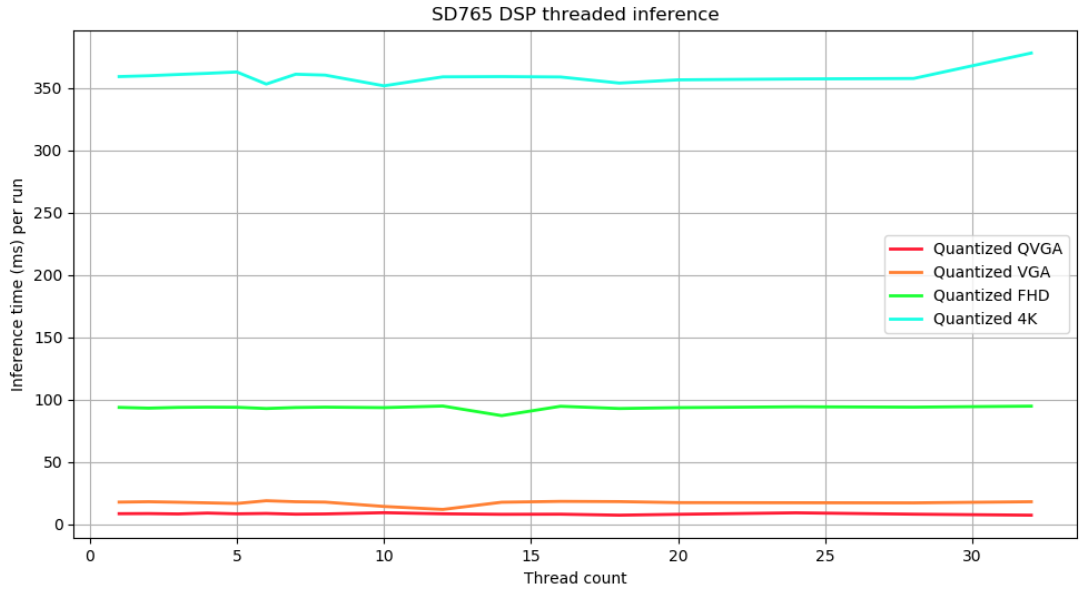

Figure 2. SD765 DSP optimal threading

Examples of noisy and biased measurement behaviours are observed. Baseline behaviour with active wireless connection is visualized in Figure 1. In a controlled situation we want to stabilize the baseline behaviour, but also have as little external activities (apps, background apps, listeners, etc.) as possible.

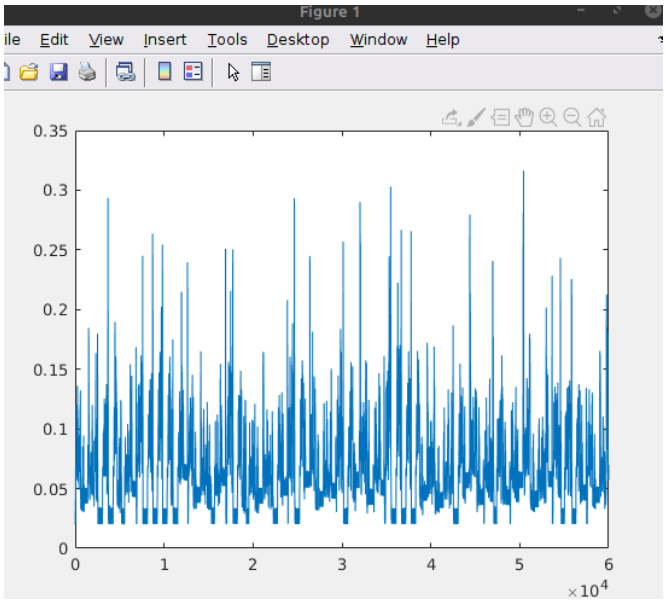

Figure 1. Baseline behaviour on active wireless connection

An USB connection is established between a PC and the test device in Figure 2. The device draws power from the USB connector, which results in biased and smaller energy consumption drawn from the measured power cable.

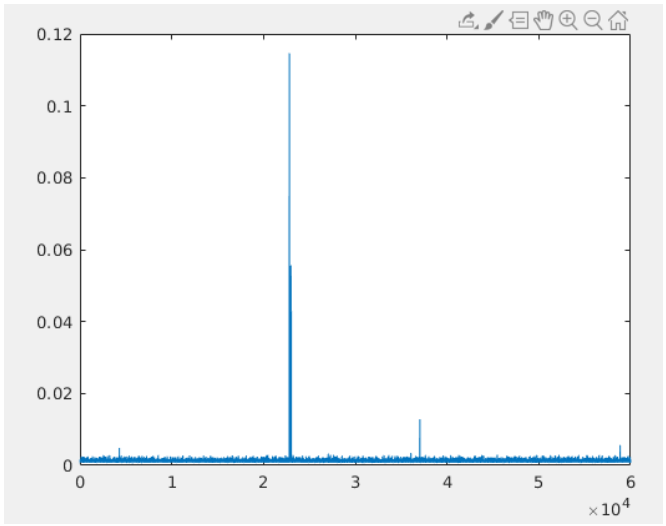

Figure 2. Baseline behaviour on active USB connection

Conducted raw power dissipation measurement visualizations, on SD865, are observed. Figure 1 displays raw FX-8 model measurements on the CPU.

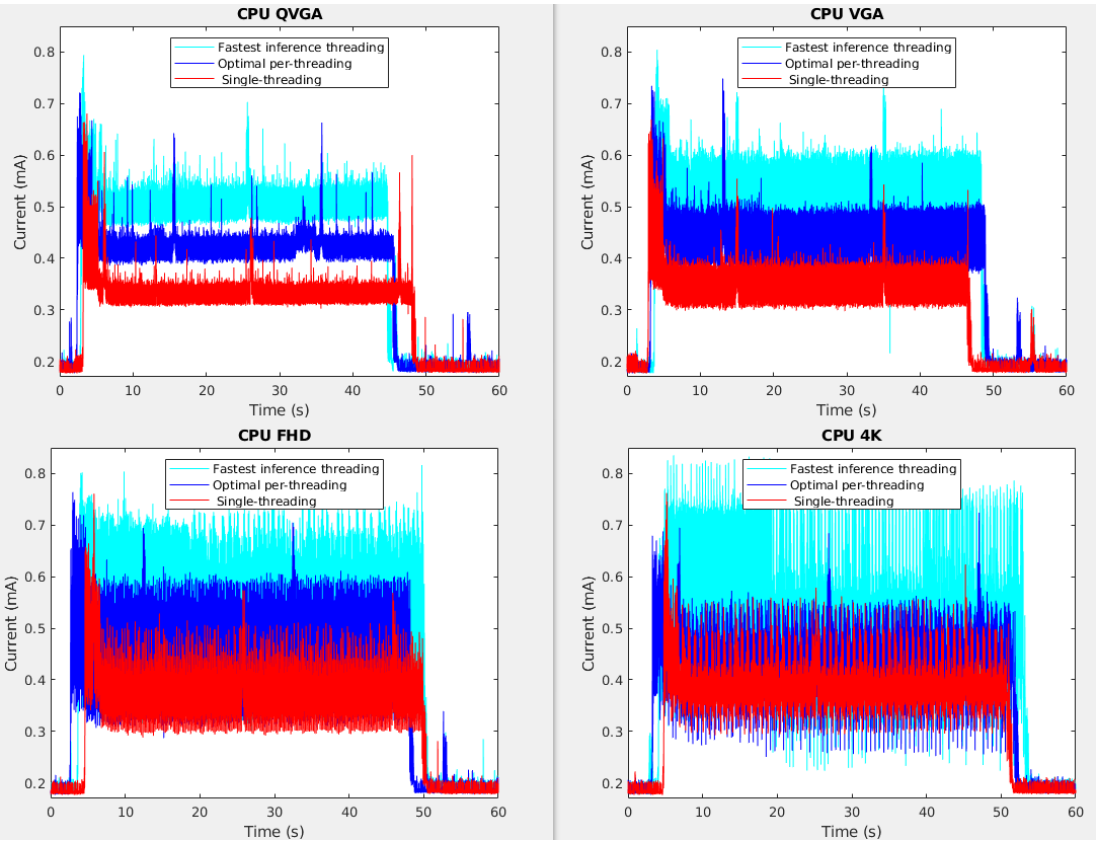

Figure 1. SD865 CPU Power Consumption by resolution

Raw power dissipation measurements for GPU FP-16 and FP-32 models, and DSP FX-8 model, are displayed in Figure 2 .

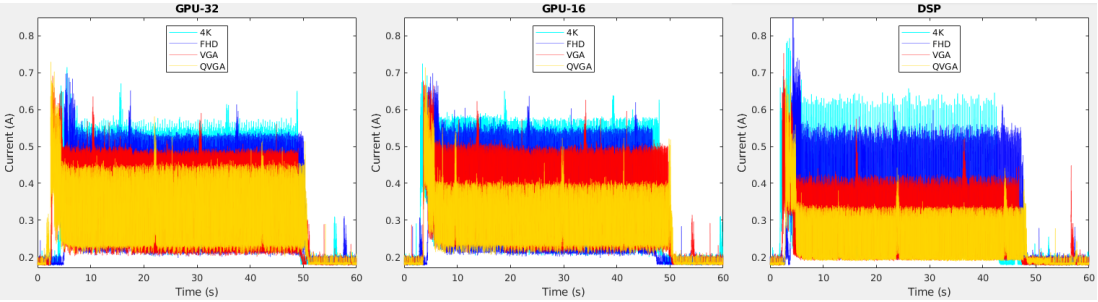

Figure 2. SD865 GPU and DSP Power Consumption by raw input mode

Power dissipation analysis setup for SD865 is observed. Table 1 contains total duration for each benchmarked inference duration.

Table 1. Measured inference total duration

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	45.004 s	44.022 s	45.197 s	48.894 s
CPU FX-8 TB	43.426 s	45.715 s	49.114 s	44.658 s
CPU FX-8 TT	41.866 s	43.933 s	46.047 s	47.879 s
GPU FP-32	52.054 s	50.591 s	49.823 s	47.163 s
GPU FP-16	50.147 s	49.894 s	45.772 s	48.684 s
DSP FX-8	48.547 s	48.092 s	47.055 s	43.538 s

Measured inference average durations for the warmup are collected to Table 2. Minimum duration of 0.5 seconds was used for warmup. Warmup duration is used for mainly removing the first couple (unstable) runs which would substantially affect the benchmarking measurement stability.

Table 2. Measured inference average during warmup

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	15.3 ms	34.5 ms	260.6 ms	1123.7 ms
CPU FX-8 TB	11.4 ms	36.5 ms	187.2 ms	740.8 ms
CPU FX-8 TT	7.0 ms	24.6 ms	154.2 ms	617.1 ms
GPU FP-32	8.5 ms	17.2 ms	111.4 ms	360.8 ms
GPU FP-16	9.0 ms	16.0 ms	111.9 ms	261.7 ms
DSP FX-8	12.1 ms	19.2 ms	78.4 ms	311.0 ms

The total inference count for each test case is categorized in Table 3.

Table 3. Measured inferences per test

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	7500	1867	248	56
CPU FX-8 TB	9783	2694	361	81
CPU FX-8 TT	11538	3309	448	110
GPU FP-32	3830	1722	357	99
GPU FP-16	4500	2642	480	160
DSP FX-8	4000	2500	700	140

For measuring the energy amount, the mean value is calculated from actual measurement range. The subtracted baseline calculations are in Table 4.

Table 4. Active mean consumption baseline subtraction

Run mode	QVGA	VGA	FHD	4K
CPU FX-8 T1	147.7 mA	161.9 mA	188.4 mA	182.9 mA
CPU FX-8 TB	238.5 mA	256.0 mA	274.5 mA	254.5 mA
CPU FX-8 TT	320.8 mA	338.7 mA	334.6 mA	345.5 mA
GPU FP-32	93.1 mA	129.0 mA	180.3 mA	213.7 mA
GPU FP-16	82.2 mA	128.3 mA	156.9 mA	195.2 mA
DSP FX-8	38.2 mA	57.8 mA	102.0 mA	123.8 mA