Exploration of a Dynamic Approximate Multiplier for Mixed-Precision Inference

Eivind Aksel Weidemann
ei0822we-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu

Examiner: Erik Larsson

March 11, 2026

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The increasing demand for efficient neural network (NN) inference on edge devices has driven the need for hardware-level optimizations that balance computational accuracy with energy and area efficiency. This thesis explores the design and implementation of a dynamic approximate multiplier capable of mixed-precision inference, focusing on floating-point (FP) formats. Using a logarithmic-approximation approach, we implement three precision modes—High Precision Correction (HPC), Low Precision Correction (LPC), and No Correction (NC)—and evaluate their performance on Artix-7 FPGA hardware.

Our results indicate that at 8-bit widths, the numerical error difference between full precision and approximate modes gives a manageable drop off in NN image classification accuracy. Hardware synthesis shows that 8-bit NC and LPC multipliers provide significant advantages in area (number of LUTs) and latency compared to commonly used 8-bit fixed-point multipliers. Furthermore, we demonstrate that a mixed-precision strategy, guided by layer-wise sensitivity, allows for maximizing hardware efficiency while maintaining nearly baseline accuracy. We conclude that approximate 8-bit floating-point multipliers are in some cases a viable alternative to fixed-point arithmetic.

Popular Science Summary

Artificial Intelligence is everywhere, but the "brains" (Neural Networks) behind it are power-hungry. To make AI run faster and with better battery life on your phone or small devices, a bit of mathematical perfection can be sacrificed. This is called Approximate Computing.

This thesis takes a look at how one of the most common tasks in AI computing can be simplified: multiplication. Normally, computers try to be 100% accurate. However, AI is surprisingly resilient; it can still "recognize" a cat even if the internal math is somewhat off. In this work, a multiplier is designed that can switch between different levels of "accuracy" on the fly.

By using "smaller" numbers in a physical sense and adjustable approximate math, a design is implemented that is cheaper and less power-hungry than other alternatives.

Table of Contents

1 Introduction **1**
- 1.1 Thesis goal . . . 1
- 1.2 Tools . . . 2
- 1.3 Hardware for Calculating Inference . . . 2
- 1.4 Previous Work . . . 3
- 1.5 Contributions . . . 4

2 Background **7**
- 2.1 Floating Point Approximate Multiplication . . . 7
- 2.2 Quantization . . . 10
- 2.3 Field Programmable Gate Arrays (FPGAs) . . . 11

3 Implementation **13**
- 3.1 Multiplier Behavioral Implementations . . . 13
- 3.2 Inference Accuracy Testing . . . 14
- 3.3 Hardware . . . 16

4 Results and Discussion **19**
- 4.1 Numerical Error Analysis . . . 19
- 4.2 Inference Accuracy . . . 19
- 4.3 Hardware Metrics . . . 20
- 4.4 Mixed-Precision and Layer-wise Impact . . . 22

5 Conclusions **27**
- 5.1 Hardware-Accuracy Trade-offs . . . 27
- 5.2 Applying Mixed-Precision to Neural Networks . . . 28
- 5.3 Designing Neural Networks for Multiple Precision . . . 28
- 5.4 Improvements and Further Research . . . 29

References **31**

A Code **33**

List of Figures

2.1 A logarithmic-approximation multiplier assumes $log_2(1+x)$ is similar to x 9

3.1 Block schematic for scheme 1 15
3.2 Block schematic for scheme 2 16
3.3 Block schematic for the implemented multiplier 17

4.1 Error heatmaps, input exponent is constant, mantissa is between 0 and 1. Error scale is equal for 16-bit multipliers and for 8-bit multipliers. 25

List of Tables

4.1 How the multiplication using different approximation modes and bit-sizes compares to full precision 32-bit floating point 20
4.2 Inference accuracy of different configurations. 21
4.3 LUT = Look-up table. FF = flip-flops. 21
4.4 LUT = Look-up table. FF = flip-flops. 21
4.5 Layer-wise Impact of Quantization and approximation on the small CNN. Approximation and quantization is only applied to a single layer 22
4.6 Layer-wise Impact of Quantization and approximation on the medium CNN. Approximation and quantization is only applied to a single layer 22
4.7 Layer-wise Impact of Quantization and approximation on ResNet9. Baseline FP32 accuracy = 86.81%. Total parameters = 42.7K 23
4.8 Layer-wise Impact of Quantization and approximation on MobileNetV2. Baseline FP32 accuracy = 91.95%. Total parameters = 2.2M 23
4.9 Accuracy gain from using LPC8 instead of NC8 above certain thresholds 23
4.10 Accuracy gain from using LPC8 instead of NC8 above certain thresholds 24

Chapter 1
Introduction

The impact of hardware optimizations for large neural networks (NNs) is substantial, and the use of dedicated neural processing units (NPUs) is spreading. NPUs implement specialized hardware designs that compute AI and machine learning applications as efficiently as possible. One common hardware optimization strategy used is approximate computing, a technique that reduces hardware costs (such as area, timing and power) at the expense of inaccurate results. This thesis will explore the accuracy impact of two ways of reducing costs of AI hardware; smaller number formats, and approximate multiplication. More specifically, this thesis will focus on mixed-precision inference, meaning number formats and/or approximations are adjusted during different stages of a NN forward pass.

1.1 Thesis goal

Broadly, the objective will be to test NNs with mixed-precision during forward passes and get statistics about the trade-off between inference accuracy and hardware metrics like area, and timing.

One goal of the thesis is to find out the possible hardware and precision metrics of an implemented mixed-precision design. This will include an analysis of how much of an impact the different precisions have on NN inference accuracy. The results will be compared to, and with, relevant data formats and full-precision multipliers.

Another goal is examining how much merit there is in being able to alter the precision during runtime. There are three sides to this goal. One is examining the span of the results, how much room there is for adjustments between the lowest and the highest precision results. A bad result would be if there is negligible difference between the most and least precise modes. Another side is exploring how to deploy mixed-precision in way that minimizes both accuracy loss while maximizing the use of low-precision. The last side is if the trade off between the costs and accuracy is worthwhile.

A final point of interest is if there is certain neural network designs or design choices that lends itself particularly well to mixed-precision hardware. Are there some neural network traits that should be avoided or leaned into.

There are two main challenges for these goals. One is hardware support, approximation techniques and different number formats generally requires dedicated

hardware to achieve the best results. The other is precision deployment, utilizing mixed-precision efficiently requires some insights into what calculations in a complex and hard to generalize NN forward pass are important and not.

1.1.1 Scope

The scope of NNs and approximation techniques is quite expansive so some limitations have been made. Regarding NNs, this thesis is limited to small and medium image classification models, specifically a couple simple convolutional neural networks (CNNs), and ResNet9 and 44. For approximation techniques, one approximate multiplier design with three precision modes will be tested [1]. The number formats that will be explored are FP32 as baseline accuracy, INT8 as the industry standard for efficient inference, and BF16 and smaller floating points formats that are compatible with the approximate multiplier design.

1.2 Tools

The project will be done using a mix of Python and C simulation, the Vivado IDE, and FPGA hardware. Hardware results will be generated using Vivado for FPGA hardware. Precision and accuracy results will be generated mainly with code simulations, where essential results also confirmed on FPGA hardware.

1.2.1 Pytorch

Pytorch is a very powerful set of open source tools and libraries to train and evaluate machine learning AI. For this project it will be used to train and evaluate neural networks.

1.2.2 Vivado

The Vivado IDE is for synthesizing hardware designs for FPGAs, it is able to program FPGA boards from RTL code and generate metrics on hardware utilization, power and timing.

1.2.3 Artix-7 FPGA

A finalized design will be programmed onto an Artix-7 FPGA using a Nexys4 board for verification of essential results.

1.3 Hardware for Calculating Inference

The choice of hardware for neural network inference is typically a trade-off between flexibility, throughput, and energy efficiency. Standard central proccesing units (CPUs) offer the highest flexibility but has limited parallel processing performance, making them inefficient for the massive amounts of operations required by neural networks [2]. Graphics processing units (GPUs) provide high throughput

through massive parallelism; however, they are often power-intensive and optimized for some fixed number-formats, making them less suitable for exploration of specialized, low-bit-width approximate multipliers used in this work.

FPGAs represent a middle ground, offering a reconfigurable fabric that allows for the implementation of custom hardware architectures tailored to specific data formats and arithmetic logic. This project utilizes an FPGA because it enables the creation of a non-standard dynamic multiplier that can switch precision modes at the gate level. Furthermore, FPGAs allow for precise measurement of hardware metrics such as Look-Up Table (LUT) utilization and path latency, which are critical for evaluating the trade-offs of approximate computing.

1.4 Previous Work

Research most relevant to this project is about the error-resilience of neural networks, quantization for efficient inference, the development of alternative number formats, and approximate arithmetic hardware.

1.4.1 Error Resilience of Neural Networks

Neural networks are inherently error-resilient, this resilience stems from the high redundancy of parameters, distribution of "knowledge" across a whole network, the regularization of activation layers, and more continuous in stead of discrete results. A popular area of research is about techniques for computing neural networks more efficiently by utilizing these factors.

Neural networks are to a high degree tolerant to intermediate errors and compressions in internal calculations [3]. Many different methods for exploiting this has been proposed, where especially quantization is all but mandatory for efficient inference [4]. Pruning, the process of removing insignificant parameters from trained networks, is another common techniques[5], but is not explored in this project. Different layers have different tolerances for accuracy drops [6]. This allows efficient designs by tailoring each layer's precision [7].

1.4.2 Quantization

In neural networks, quantization is the process of mapping high-bit representations of weights and activations to lower-bit formats. This technique enables reduced model size, faster inference, and lower power consumption, at the expense of a reduction in result accuracy. Quantization is often categorized based on when the quantization occurs, post-training quantization (PTQ) and quantization-aware training (QAT). Where PTQ allows quick deployment without re-training and is often sufficient down to 8-bits. For more aggressive quantization with minimal accuracy loss, QAT is generally required. Quantization can be mixed-precision, where sensitive layers are given higher bit-widths. In this project, PTQ is used with fixed bit-widths.

Number format

The numbers used in machine learning has continually been pushed to smaller formats, achieving significant computational gains with minimal degradation in accuracy.

The go-to approach when computing inference on edge devices is to use 8-bit integers. The fixed-point format is inherently simpler in hardware and an 8-bit format gives in most cases good gains in terms of hardware costs with very little drop in result accuracy. Fixed-point arithmetic uses less power than area than float, also at smaller bit-widths like 8 bits [8]. The comparison between fixed and float is more nuanced when using approximate floating-point multiplication.

Purely considering result accuracy, the significant difference between fixed and float is the ability to capture outliers [8]. For layers with parameters that are significant outliers, floating-point is better for capturing these. This can especially be a problem in very large language models, which might need special techniques to handle outliers [9].

Still, the best incentive for using floating-point over fixed-point inference is the elimination of the complex quantization step.

Approximate Computation

There are multiple designs for approximate multipliers, this thesis will explore a 16-bit floating-point multiplier with three different approximation modes [1]. The modes are high-precision correction, low-precision correction, and no correction. The mantissa operation is approximated and the approximate result is compensated with an adjustable error-correction module. The exponent operation is always exact and not approximated to any degree.

1.5 Contributions

This thesis provides a analysis of the interaction between numerical compression and arithmetic approximation. The primary contributions of this work are as follows:

- **Framework for Combined Optimization:** This work proposes a method for integrating quantization with mixed-precision arithmetic approximation. This exploration addresses whether these two strategies target the same underlying error-resilience of neural networks and if their combined use offers additive benefits in hardware efficiency without compounding accuracy loss.

- **Dynamic Hardware Design:** An implementation of a runtime-reconfigurable multiplier design. Unlike static approximate designs, this architecture provides the user with the flexibility to dynamically trade inference accuracy for energy savings and increased speed, depending on the specific requirements of the application or the sensitivity of the neural network layer currently being processed.

- **Comparative Analysis of FP8 vs. INT8:** This work contributes a detailed comparison between the industry-standard INT8 fixed-point inference and an approximate FP8 floating-point approach.

Chapter 2
Background

This chapter establishes the theoretical framework necessary to understand the design and evaluation of the dynamic approximate multiplier and the quantization. It begins with an overview of floating-point arithmetic and the mathematical principles behind logarithmic-approximation multiplication. The chapter explores quantization techniques for both fixed-point and floating-point formats, which are essential for reducing the computational footprint of neural networks. Finally, a short overview of Field Programmable Gate Array (FPGA) architecture and programming is provided.

2.1 Floating Point Approximate Multiplication

2.1.1 Floating Points

The IEEE 754 standard used for floating-point computation in almost every modern processor defines the value of a number A as:

$$A = (-1)^{sign} \times (1 + mantissa) \times 2^{exponent-bias}$$

Every IEEE 754 number is represented by three sets of bits: sign, exponent, and mantissa. The sign indicates if the number is positive or negative. The exponent bits to determine how far the decimal (binary) point shifts. The mantissa represents a fractional number between zero and less than one. The mantissa has an implicit leading bit of 1 for normalized numbers, when the exponent is zero, a floating-point is subnormal and the leading bit is removed, but in this project all subnormal numbers are simplified to zero.

For a more intuitive explanation of the exponent and mantissa, one can say that the exponent decides between which two exponents a number is between and the mantissa decides the point between those two. For example, with an exponent of 4, the number is between 2^4 and 2^5. Then, if the mantissa is 0.5 the number is right in the middle. Calculated with 1.5×2^4.

This example omits the bias which is used for representing negative exponents without the need for a separate sign bit for the exponent. In the single-precision floating-point format (FP32), the exponent has the range $[0, 255]$ and the bias is 127, so a number between 2^4 and 2^5 would use the exponent $4 + 127 = 131$ and an exponent of 4 would be a number between 2^{-123} and 2^{-124}. The bias is not stored

with the number and arithmetic with floating points has to make assumptions about what the bias is. For this reason, multipliers usually only allows operation between equal formats where the bias is the same. To simplify further expressions, s denotes sign, m denotes mantissa, and e denotes (exponent-bias).

2.1.2 Multiplication

When multiplying two floating-points, there are three principal operations: exclusive or of the signs, addition of the exponents, and multiplication of the mantissas.

$$A \times B = (-1)^{s_A \oplus s_B} \times ((1 + m_A) \times (1 + m_B)) \times 2^{e_A + e_B} \tag{2.1}$$

In actual computing, there are additional operations that ensures the result conforms to a standard representable format. These operations are normalizing, rounding, and checking for exceptions. In an approximate multiplier, rounding and exception checking can be simplified for small improvements in area and timing slack, but the operation with the most potential gain from simplifying is the mantissa multiplication.

One well-documented way of doing approximate multipliers is using logarithm-approximation, based on the property $\log(A \times B) = \log(A) + \log(B)$. This is often referred to as Mitchell's Algorithm, and using this, the hardware costs of a floating-point multiplier can be significantly reduced.

The absolute value of a floating-point can be calculated as:

$$|A| = (1 + m) \times 2^e$$

There is a logarithmic approximate relationship (Figure 2.1):

$$\log_2(1 + x) \approx x \quad \text{or} \quad 1 + x \approx 2^x \quad \text{for } 0 \leq x \leq 1$$

Because the mantissa represents a number between 0 and 1, the logarithmic value of a floating point can be approximated with:

$$\log_2(|A|) = \log_2(1 + m) + e \approx m + e$$

Now the approximate product can be expressed as:

$$\begin{aligned} \log_2(|A \times B|) &= \log_2(|A|) + \log_2(|B|) \\ &\approx e_A + e_B + m_A + m_B \end{aligned}$$

Taking the inverse transformation:

$$\begin{aligned} |A \times B| &\approx 2^{e_A + e_B + m_A + m_B} \\ A \times B &\approx (-1)^{s_A \oplus s_B} \times 2^{e_A + e_B + m_A + m_B} \end{aligned}$$

Written as a IEEE 754 number with the implicit 1 and considering the case when the mantissa overflows, the expression is:

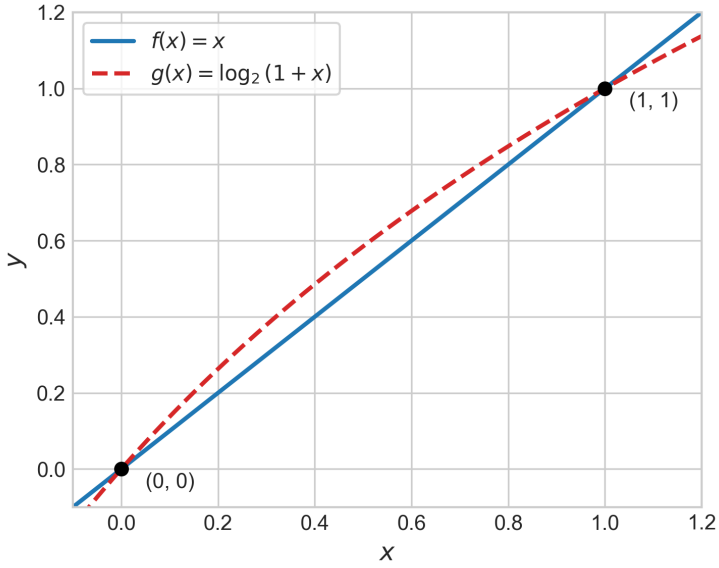

Figure 2.1: A logarithmic-approximation multiplier assumes $log_2(1+x)$ is similar to x

$$A \times B \approx (-1)^{s_A \oplus s_B} \times \begin{cases} (1 + m_A + m_B) \times 2^{e_A + e_B} & m_A + m_B < 1 \\ (1 + (m_A + m_B - 1)) \times 2^{e_A + e_B + 1} & m_A + m_B \geq 1 \end{cases} \quad (2.2)$$

In terms of hardware costs, equation 2.2 is a significant improvement from equation 2.1 because the mantissa multiplication is changed to addition. A 24×24 bit multiplier needed for single precision floating-points can be changed to a 24-bit adder instead. Somewhat simplified, the hardware cost of an adder scales linearly with bit-size, while a multiplier scales quadratically given that multiplication is in essence repeated addition.

2.1.3 Error Correction

Using logarithmic-approximation, the product will habitually be lower than the actual value. Figure 2.1 shows why, x is smaller than $log_2(1+x)$ between zero and one. To rectify this it is possible to add error correction[1].

Expanding equation 2.1 and considering the case when the mantissa overflows:

$$A \times B = (-1)^{s_A \oplus s_B} \times \begin{cases} (1 + m_A + m_B + m_A m_B) & m_A + m_B + m_A m_B < 1 \\ \quad \times 2^{e_A + e_B} & \\ \dfrac{1 + m_A + m_B + m_A m_B}{2} & m_A + m_B + m_A m_B \geq 1 \\ \quad \times 2^{e_A + e_B + 1} & \end{cases} \tag{2.3}$$

Shows that the error of the approximate equation 2.2 is the product of m_A and m_B and a division by two (a simple bit-shift in hardware) when the mantissa overflows. The proposed error correction is to calculate $m_A m_B$ with multiplication, but only using the most significant bits. Depending on how many bits are used in this multiplication, the trade-off between precision and hardware costs can be adjusted. With smart hardware design, it is possible to adjust this trade-off dynamically.

2.2 Quantization

Quantization is the process of mapping continuous, high-precision, or large-set values to a smaller, discrete set of finite values.

2.2.1 Quantizing Floating-Points to Fixed-Points

For neural networks, the idea is to take a set of trained weights and biases, quantize them to fixed-points with smaller bit-widths, evaluate with these easier to compute numbers, and finally de-quantize the result back to the real value. This is one of the most impactful ways to decrease hardware costs and often with minimal loss of precision.

There are many ways to do quantization, with a wide array of design choices when selecting quantization scheme, granularity and bit-width[4]. Determining a best strategy requires comprehensive practical considerations. A simple, but robust and often a default option is per-tensor symmetric uniform quantization. This scheme is defined by a *scale factor* s and a *bit width* b and applied per-tensor/layer-wise in a neural network. Using this the quantization step from a value x to signed integer x_{int} is defined as:

$$x_{int} = clamp(\lfloor \frac{x}{s} \rceil; -2^{b-1}, 2^{b-1} - 1)$$

Where $\lfloor \cdot \rceil$ is the round-to-nearest operator and clamping is defined as:

$$clamp(x; a, c) = \begin{cases} a, & x < a, \\ x, & a \leq x \leq c, \\ c, & x > c. \end{cases}$$

And the de-quantization step to approximate the real-valued input x:

$$x \approx \hat{x} = s(x_{int})$$

With this approach, quantizing the array [-2.60, 0.00, 0.44, 44.00] to INT8, using the scaling factor 44/127 to utilize the whole range of INT8, the quantized array would be [-8, 0, 1, 127]. After de-quantization the approximate original array would be [-2.77, 0.00, 0.35, 44.00]. Scaling allows a distribution between -1 and 1 to be represented with fixed points.

2.2.2 Quantizing Floating-Points to Smaller Floating-Points

Quantizing floating-points differ based on if the exponent bit-width is changed or not.

Downsizing floating-point formats without changing the exponent bit-width can be done by simply truncating mantissa bits to decrease precision. This is one advantage of floating-points, "Quantization" can be done by just a simple cast operation. Scaling is also not necessary because the precision is relative to the size of the number. Whether a number is 1.0 or 1000.0, there still is only a set amount of bits of significant digits to describe it.

When quantizing to a new exponent bit-width, the inherent bias needs to be adjusted and scaling should be used to ensure that the distribution of the original values is shifted into the range of the new format.

For example, if there is a distribution of very small values that risks being quantized to zero without scaling. Then if there is a lot of unused space for higher numbers, it would be beneficial to scale up the distribution to utilize the full range. In practice, shifting up the lower mantissa bits that would be truncated, into the more significant spots and even into the exponent bits.

2.3 Field Programmable Gate Arrays (FPGAs)

Unlike a standard CPU or GPU, which possesses a fixed hardware architecture designed to execute sequential streams of instructions, an FPGA is an integrated circuit designed to be configured by the designer after manufacturing.

An FPGA typically consists of three primary components:

- **Look-Up Tables (LUTs):** A LUT, or a set of LUTs, can emulate combinatorial logic by storing truth tables.
- **Flip-Flops:** These provide memory or 'state' for synchronous logic, allowing FPGAs to handle sequential operations and timing.
- **Programmable Interconnects:** A web of routing channels that allows the output from any logic block to be connected to the input of another.

To "program" an FPGA, hardware description languages are used to describe the physical structure and behavior of a digital circuit. This describes the LUTs and the routing between them, making it possible to transform the FPGA into anything from a simple timer to a parallel processor for neural networks with tailored multipliers and number-formats.

Chapter 3
Implementation

The list of implementations in this project include:

- A behavioral simulation of the approximate multiplier in C.
- A neural network accuracy testbench supporting configurable number format, multiplier precision, and quantization strategy across multiple models.
- An RTL implementation of the multiplier with dynamic mixed precision on an FPGA.

3.1 Multiplier Behavioral Implementations

During testing there are a few multiplication scenarios with different precision modes and number formats that needs to be handled correctly.

For integer numbers, switching between, and multiplying different bit-formats is trivial and do not require any special implementations. For the purposes of this implementation it is sufficient to just always use 32-bit integer and the default '*'-operator, smaller bit-widths can be simulated by simply bounding larger formats to a smaller range, for example between -128 and 127 to simulate an 8-bit integer.

For floating points, it is not as straightforward. Both switching between and multiplying different bit-widths require some bit manipulation to function properly. When up- or downsizing floating point formats, the simple, but not quite correct, method used in this implementation, is to truncate and map bits to the correct position without any mantissa rounding or exponent re-biasing.

This method introduces two inaccuracies when downsizing floating points, first, the truncation will always round down, the worst case when mapping from a 32-bit floating point to a 8-bit floating point with 3 mantissa bits is an error of $1/2^3$. For example, 2.2499_{fp32} will be truncated to 2.0_{fp8}. With proper rounding the worst case would only be half as bad.

The second inaccuracy relates to underflow and appears when converting numbers close to zero to a format with fewer exponent bits. To simplify the underflow handling, if the extracted exponent is zero, the mantissa is also set to zero. For an 8-bit floating point, this means that 7 numbers between 0 and 0.015625 where the exponent is zero while the mantissa is not, is wasted. This can have an significant impact on the inference accuracy if numbers are not scaled to take advantage of the full range.

The C implementation for simulating simple downsizing of a 32-bit floating point to 8-bit is shown in listing 3.1.

```
float convert(float a) {
    union { float f; uint32_t u; } ua = { a }, ur;

    // Extract sign, exponent, mantissa
    uint32_t sign_a = ua.u >> 31;
    uint32_t exp_a  = (ua.u >> 23) & 0xFF;
    uint32_t man_a  = ua.u & 0x700000; // Truncate

    // Simplified underflow
    if (exp_a <= 120)
        return 0.0f;

    // Overflow
    if (exp_a > 135)
        exp_a = 135;

    ur.u = (sign_a << 31) | (exp_a << 23) | man_a;

    return ur.f;

}
```

Listing 3.1: Simple downsizing from 32-bit to 8-bit

This approach allows for using the default '*'-operator when multiplying with full precision, as long as the result is checked for under- and overflow and the mantissa is truncated.

The approximate floating point multipliers are implemented as custom functions and shown in Appendix A, (see listing A.1). There are three different approximation modes, in order of decreasing accuracy: high-precision correction (HPC), low-precision correction (LPC), and no correction (NC). NC only sums the mantissas, LPC sums the mantissas and adds the product of the three most significant bits, and HPC sums the mantissas and adds the product of the six most significant bits. These multipliers are tested as both 8-bit and 16-bit variants, referred to as HPC8 and HPC16 respectively.

3.2 Inference Accuracy Testing

The accuracy testing is done with image classification of the CIFAR10 dataset [10]. When testing the accuracy, the complete list of variables is:

Models: Two simple CNNs of different sizes (Listing A.2, A.3), ResNet9 [11], MobileNetV2.

Formats: FP32, INT8, BF16, FP8

Multiplication precisions: Full precision, three approximate methods for floating point formats.

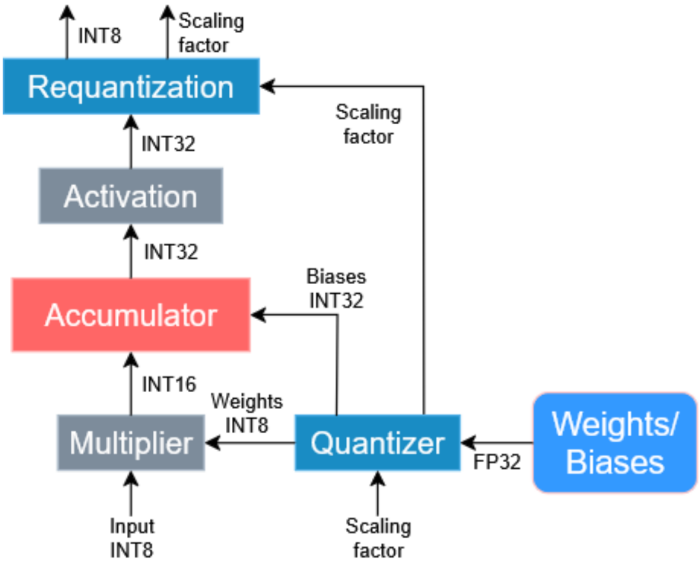

Figure 3.1: Block schematic for scheme 1

Quantization strategies: No quantization, scheme 1, scheme2.

In addition, some variables can be changed during the forward pass, which gives a lot of different combinations to test.

For most of these combinations there is little need to test them particularly exhaustive, they server more to give some perspective for the more interesting accuracy results. The most competitive and relevant comparison for this project, is the comparison between INT8 with full quantization and the approximate FP8 with quantization limited to weights only.

For mixed-precision schemes, the testing is limited to one number format and multiplication precision for each layer.

3.2.1 Quantization Schemes

There are two different quantization schemes simulated. One intended to work best for fixed-points, and another intended for floating-points multiplication with the approximate multiplier, called *Scheme 1* and *Scheme 2* in figure: 3.1 and 3.2 respectively. Compared to the proposed design for the approximate multiplier, which is for multiplication with two 16-bit numbers, scheme 2 is for multiplication between one 8-bit and one 16-bit. This allows weights to be stored with 8-bits.

The quantization method used in scheme 1 is symmetric uniform quantization, as described in Chapter 2, with re-quantization between each layer. Both weights and inter-layer intermediate results is stored with 8-bits. When using this quantization scheme, the fixed points will overflow during the accumulation, unless a wider bit-width is used for accumulation. In this scheme, the multiplier is 8×8

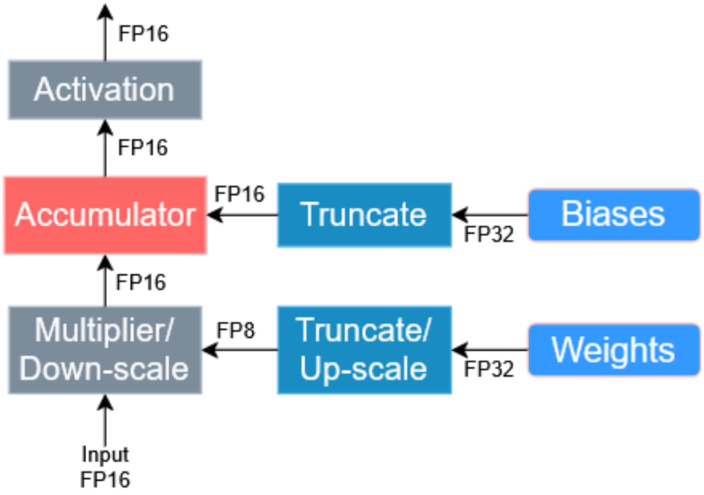

Figure 3.2: Block schematic for scheme 2

bits, and the accumulator is 32 bits. Using this, the results from a layer will be 32 bits and needs to be re-quantized back down to 8 bits before being passed to the next layer. To re-quantize the numbers correctly, the prior scaling factors has to be known, usually these factors are fixed and calculated during training of a network. The alternative method, calculating scaling factors on the fly, poses significant optimization challenges for parallel computing. Calculating the scaling factors beforehand requires that there is representative test data for the network available.

For the scaling of floating point numbers (scheme 2), the scaling factor is restricted to a power of 2 and there is no re-quantizing between layers. This means the down-scaling needs to happen before accumulation, which can be achieved with a simple bias-shift in a floating-point multiplier. Weights is stored with 8-bits, and inter-layer intermediate results is stored with 16-bits. Because of the wider range of floating-points combined with using 16-bits for intermediate results, this scheme does not run the same risk of overflowing during accumulation, and for this reason re-quantization can be omitted between layers. In this scheme the multiplier is 8×16 bits and the accumulator is $16 + 8$ bits.

3.3 Hardware

The implemented multiplier on hardware has some changes from the proposed multiplier design[1]. The most significant change is that instead of two 16-bit inputs there is one 8-bit and one 16-bit input. Because the weight is 8-bit, the HPC mode is also omitted, as the 8-bit floating-point format used has only 3 mantissa bits. This makes the HPC mode almost identical to the LPC mode in testing because the LPC mode already uses all the available bits for the error correction.

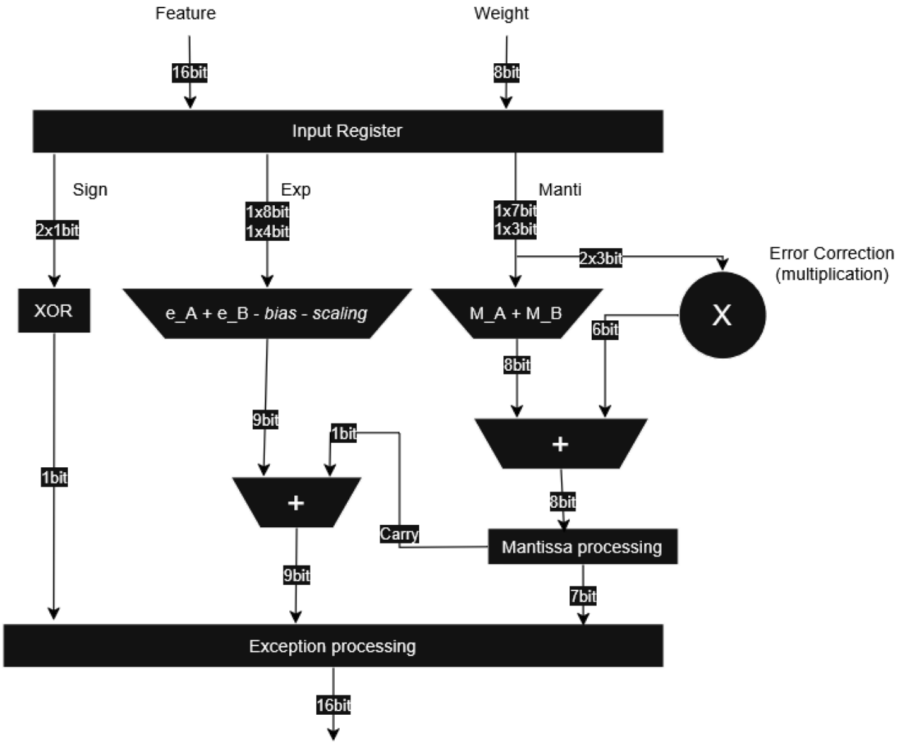

Figure 3.3: Block schematic for the implemented multiplier

Another change is that the multiplier also supports a up- or down-scaling of numbers, because the scaling factor in scheme 2 is restricted to a power of 2, the back-scaling is simply done using subtraction/addition in the exponent operation. The block schematic is shown in figure: 3.3. The accuracy results in the next chapter is not generated on the implemented hardware, but the hardware is confirmed to produce the same results as the behavioral C implementation for both multiplications, and a small sample of image classifications using a previously implemented FPGA CNN image classifier[12].

Chapter 4
Results and Discussion

This chapter presents the experimental findings of the thesis, evaluating the proposed dynamic approximate multiplier across numerical, architectural, and hardware-level metrics. The analysis begins with an overview of the numerical error introduced by the different approximation modes, followed by an evaluation of their impact on the inference accuracy of Convolutional Neural Networks (CNNs). Finally, the hardware utilization and timing results obtained from FPGA synthesis are presented to quantify the efficiency-accuracy trade-off.

4.1 Numerical Error Analysis

The numerical error of the High Precision Correction (HPC), Low Precision Correction (LPC), and No Correction (NC) modes using both 8-bit and 16-bit was evaluated through a multiplication sweep of randomly selected FP32 pairs between 1.0 and 2.0. The resulting errors for both 16-bit and 8-bit formats are summarized in Table: 4.1. The error is calculated by comparing the approximate product with the full-precision (FP) 32-bit product.

The results shows that the relative error difference between full precision and approximate modes decreases as the bit-width is reduced. Notably, at 8 bits, the difference between full precision, HPC, and LPC becomes negligible for practical applications, as the quantization error inherent in the 8-bit format begins to dominate over the arithmetic approximation error, and the LPC mode uses all three available mantissa bits to approximate the product, HPC and FP requires more mantissa bits to perform meaningfully better than LPC. This is further visualized with absolute-error heatmaps (Figure 4.1), which illustrate the distribution of error across the number combinations.

4.2 Inference Accuracy

The impact of the different parameters was tested using two representative CNN architectures on the CIFAR-10 dataset. The configurations includes various number formats and two different quantization schemes, one more fit for fixed-point and the other more fit for floating-point. There are some interesting things to note from the results presented in table 4.2.

Precision	Max Relative Error	Mean Relative Error
FP32	0.0%	0.0%
FP16	1.0%	0.2%
HPC16	2.2%	1.0%
LPC16	5.8%	2.0%
NC16	11.5%	3.9%
FP8	14.4%	3.5%
HPC8	14.4%	3.5%
LPC8	15.2%	3.5%
NC8	18.1%	4.8%

Table 4.1: How the multiplication using different approximation modes and bit-sizes compares to full precision 32-bit floating point

- **Baseline Performance:** The FP32 baseline has an accuracies of 71.89% and 88.25% for CNN1 and CNN2 respectively. Full-precision INT8 with requantization between layers (Scheme 1), has a negligible drop-off in accuracy.
- **8-bit Performance:** There is little accuracy loss when moving to 8-bit formats, making it difficult to justify higher bit-widths.
- **Comparison:** The most interesting comparison (highlighted in **bold**) is **INT8 (Scheme 1)**, often the go-to configuration for inference, versus **FP8 (LPC, Scheme 2)** the approximate floating-point approach. Although slightly lower accuracy, the floating-point configuration uses the easier to implement quantization scheme 2.

4.3 Hardware Metrics

Hardware metrics was measured through Vivado synthesis for an Artix-7 FPGA. The results (Table 4.3) demonstrate that the NC8 and LPC8 multipliers provide notable area and latency advantages over a standard INT8 multiplier, while HPC8 does not.

When considering a full Multiply-Accumulate (MAC) unit using the implemented 8 x 16 bit multiplier with NC and LPC (Table 4.4), the cost of adding a floating-point accumulator is notably higher than its fixed-point counterpart. HPC is omitted because there is little to no difference from LPC in terms of accuracy but substantially more expensive in terms of hardware costs. The INT8 MAC utilizes 176 LUTs, whereas the dynamic NC8 and LPC8 MAC requires 252 LUTs, and the latency is $\approx 2.9\times$. It is important to note, that this implementation uses a naive floating-point accumulator; more optimized designs, such as those using a fixed-point approach with delayed normalization, could significantly reduce the latency.

Format	Precision	Quantization	CNN1	CNN2
FP32	Full	None	71.89%	88.25%
BF16	Full	None	71.90%	88.23%
	HPC	None	71.55%	88.30%
	LPC	None	71.46%	88.53%
	NC	None	70.78%	88.00%
INT8	Full	Scheme 1	**71.75%**	**88.17%**
		Scheme 2	68.82%	87.32%
FP8	Full	Scheme 1	68.41%	87.32%
		Scheme 2	69.92%	87.84%
	HPC	Scheme 1	68.37%	87.32%
		Scheme 2	69.88%	87.86%
	LPC	Scheme 1	68.37%	87.43%
		Scheme 2	**69.78%**	**87.85%**
	NC	Scheme 1	65.51%	86.03%
		Scheme 2	68.81%	87.14%

Table 4.2: Inference accuracy of different configurations.

Multiplier Only			
	LUTs	FFs	Latency
INT8	81	51	4.677 ns
NC8	36	63	3.443 ns
LPC8	48	63	4.149 ns
HPC8	55	63	6.054 ns

Table 4.3: LUT = Look-up table. FF = flip-flops.

Multiplier and Accumulator			
	LUTs	FFs	Latency
INT8	176	149	4.784 ns
NC8 + LPC8	252	129	13.734 ns

Table 4.4: LUT = Look-up table. FF = flip-flops.

CNN1				
Layer	conv1	conv2	conv3	linear1
#Weights	432	4608	18.4K	10.2K
Quant	-1.34%	-0.74%	-0.08%	-0.42%
NC	-1.26%	-0.41%	-0.19%	-0.03%
Quant + NC	-2.33%	-0.79%	-0.48%	-0.46%

Table 4.5: Layer-wise Impact of Quantization and approximation on the small CNN. Approximation and quantization is only applied to a single layer

CNN2					
Layer	conv1	conv2	conv3	linear4	linear5
#Weights	864	17.4K	73.7K	1.05M	5120
Quant	-0.29%	-0.02%	-0.06%	+0.04%	-0.04%
NC	-0.20%	-0.05%	-0.11%	-0.10%	+0.06%
Quant + NC	-0.39%	-0.17%	+0.03%	-0.05%	-0.05%

Table 4.6: Layer-wise Impact of Quantization and approximation on the medium CNN. Approximation and quantization is only applied to a single layer

4.4 Mixed-Precision and Layer-wise Impact

To explore the viability and application of mixed-precision, individual layers were singled out and subjected to quantization and approximation (Table 4.5, 4.6, 4.7, and 4.8). The percentage drop indicates how many percentage points the accuracy drops compared with the 32-bit full precision baseline, when a specific layer is subjugated to quantization, approximation or both. The results shows that the accuracy impact varies greatly between layers. The accuracy loss from combining both approximations and quantization is generally larger than the sum of the accuracy losses from the techniques isolated, indicating that the techniques impacts each other negatively when used together. The most consistent correlation was found between layer order and accuracy loss; earlier layers are more sensitive and more exposed to precision loss than later stages of the network. A promising correlation is that big layers in terms of number of weights is not very sensitive to precision loss, meaning layers that would give the most benefit from approximations does not give a proportional loss in accuracy.

Another mixed-precision scheme that is tested is weight filtering based on magnitude (Table 4.9 and 4.10. Only weights exceeding a certain threshold uses the more precise LPC8 mode, while others used NC8.

The results shows that there is a disproportional accuracy gain from applying higher precision to individual weights with high absolute value. While magnitude

ResNet9

Layer	conv1	conv2	conv3	conv4	conv5	conv6
#Weights	864	2304	2304	2304	2304	4608
Quant	-0.45%	-0.22%	-0.11%	-0.35%	-0.33%	-0.78%
NC	-0.09%	-0.15%	-0.03%	-0.21%	-0.09%	-0.37%
Quant + NC	-0.45%	-0.84%	-0.00%	-1.01%	-0.97%	-1.80%
Layer	conv7	conv_sc	conv8	conv9	linear1	ALL
#Weights	9216	512	9216	9216	320	42.7K
Quant	+0.08%	-0.00%	-0.19%	-0.12%	-0.17%	-3.71%
NC	+0.04%	-0.04%	-0.03%	-0.14%	-0.12%	-2.07%
Quant + NC	-0.16%	-0.14%	-0.28%	-0.22%	-0.22%	-9.80%

Table 4.7: Layer-wise Impact of Quantization and approximation on ResNet9. Baseline FP32 accuracy = 86.81%. Total parameters = 42.7K

MobileNetV2

Layer	conv1	expand1	depth1	project2
#Weights	992	780K	92.4K	967K
Quant	-49.97%	-2.12%	-0.23%	-0.60%
NC	-50.90%	-0.40%	+0.05%	-0.42%
Quant + NC	-50.75%	-4.62%	-1.67%	-1.90%
Layer	depth2	project2	conv2	ALL but conv1
#Weights	416	576	415K	
Quant	-0.50%	-0.25%	+0.02%	-12.57%
NC	-0.25%	-0.05%	-0.03%	-5.75%
Quant + NC	-0.92%	-0.78%	+0.02%	-43.78%

Table 4.8: Layer-wise Impact of Quantization and approximation on MobileNetV2. Baseline FP32 accuracy = 91.95%. Total parameters = 2.2M

CNN1

Threshold	>inf	>256	>128	>64	>32	≥ 0
#Weights	0	18	316	1 701	3 754	33 712
%Weights	0%	0.05%	0.94%	5.04%	11.1%	100%
Accuracy	+0%	+0.03%	+0.29%	+0.08%	+0.93%	+0.97%

Table 4.9: Accuracy gain from using LPC8 instead of NC8 above certain thresholds

	CNN2					
Threshold	>inf	>256	>128	>64	>32	≥ 0
#Weights	0	57	1 440	26.0K	144K	1.15M
%Weights	0%	0.00%	0.12%	2.27%	12.6%	100%
Accuracy	+0%	+0.16%	+0.29%	+0.45%	+0.69%	+1.40%

Table 4.10: Accuracy gain from using LPC8 instead of NC8 above certain thresholds

filtering has some merit, comparing it with a strategy of applying higher-precision to only the first layer of each model, the first-layer strategy will generally outcompete the magnitude filtering strategy both in terms of accuracy gain and amount of high-precision multiplications. Another drawback of the magnitude filtering strategy is that it cannot take advantage of the lower latency of the NC multiplier, because it is infeasible to adjust the clock speed for single multiplications.

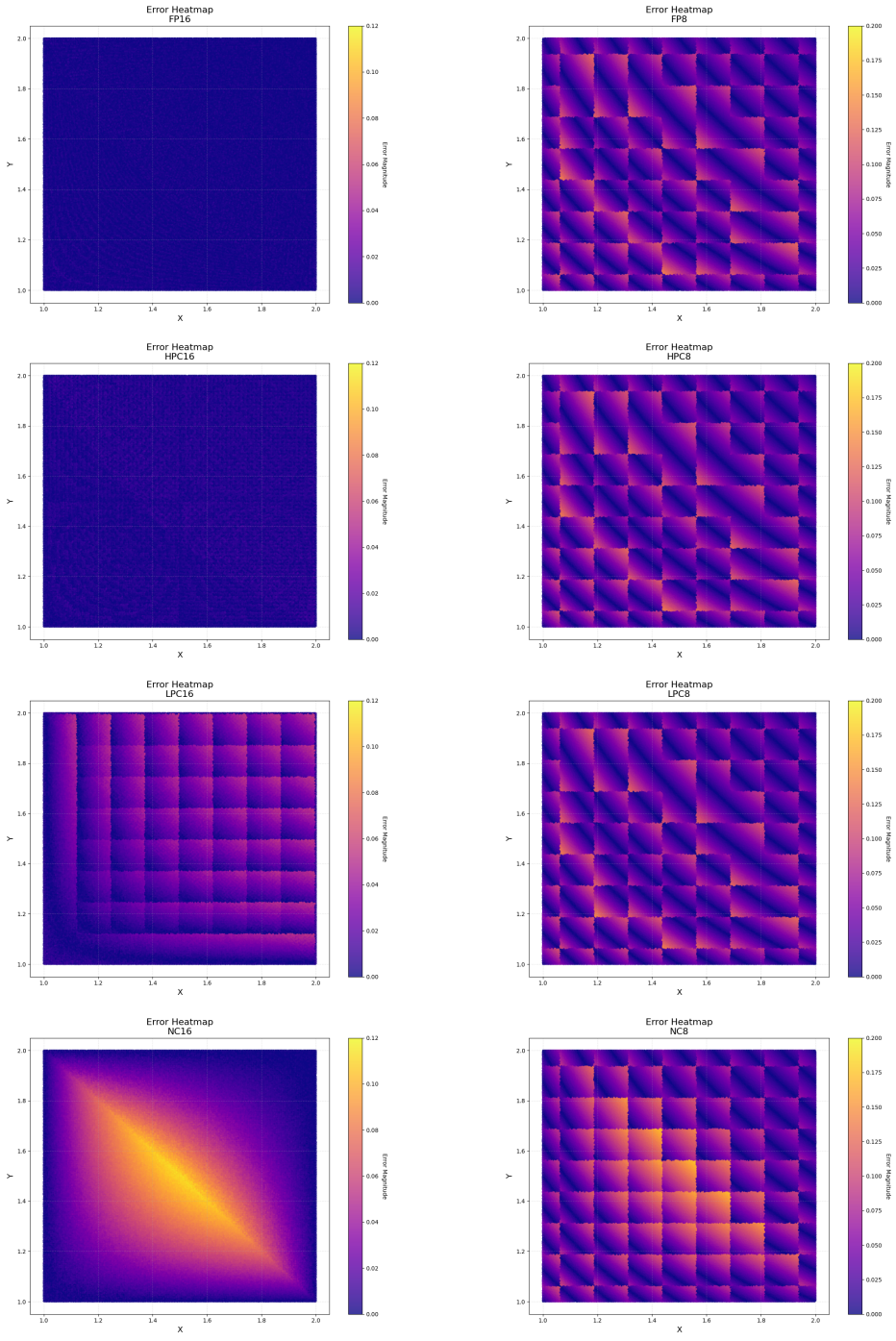

Figure 4.1: Error heatmaps, input exponent is constant, mantissa is between 0 and 1. Error scale is equal for 16-bit multipliers and for 8-bit multipliers.

Chapter 5
Conclusions

The results presented in Chapter 4 provide a overview of the trade-offs between computational precision, hardware efficiency, and neural network performance. This chapter evaluates these findings to find the viability of dynamic approximate floating-point multipliers for mixed-precision inference.

5.1 Hardware-Accuracy Trade-offs

The core objective of this thesis was to analyze the trade-off between inference accuracy and hardware metrics using approximate multiplication. The findings indicate that for the tested CNN architectures, 8-bit formats provide a highly efficient baseline with negligible accuracy loss compared to FP32. Specifically, moving from FP32 to BF16 or even 8-bit formats results in minor degradations, with some 8-bit configurations remaining within 2-3% of the baseline.

From a hardware perspective, the NC8 and LPC8 multipliers demonstrate advantages over a standard 8-bit multiplier in terms of area and latency. The NC8 multiplier, for instance, requires only 36 LUTs compared to 81 for INT8, while also offering noteworthy lower latency (3.443 ns vs 4.677 ns). However, the inclusion of a floating-point accumulator might significantly increases the overall hardware cost. While an INT8 multiply-accumulate (MAC) unit utilizes 176 LUTs, the NC8 + LPC8 MAC implementation in this work requires 252 and has significantly higher latency.

Depending on how much optimization can be done on a floating-point accumulator, an approximate floating-point MAC unit could potentially be a straight-up improvement either in latency, area, or both. In that case, this design could have broad usefulness for designs intending to maximize for those metrics. On the other hand, if the full MAC design, cannot compete on hardware metrics, it is harder to recommend using approximate floating-point arithmetic over fixed-point, unless the design needs to use floating-point arithmetic for some other reason. For example, LLMs often use floating-points to be able to handle significant outliers that might appear during inference [9].

An advantage of using floating-point arithmetic is to reduce engineering costs by making it possible to omit complex inter-layer re-quantization steps required by INT8 to avoid significant accuracy loss. This "hidden" cost of INT8, has little impact on hardware metrics, but has some costs both in terms of engineering

complexity and the need for calibration data. Still, compared to the engineering complexity of implementing a custom floating-point multiplier, the lesser engineering complexity of using a simple quantization scheme is not a solid selling point, unless lack of calibration data is a specific problem and a well-calibrated quantization sccheme cannot be implemented.

5.2 Applying Mixed-Precision to Neural Networks

The variation in layer sensitivity suggests that a "one-size-fits-all" approach to precision is suboptimal. The results show a strong correlation between layer order and error resilience, with the strongest relation being initial layers are generally more sensitive to precision loss. For example, in the small CNN, applying NC to the first convolutional layer resulted in a -1.26% accuracy drop, whereas applying it to the final linear layer had a negligible impact of -0.03%. It is also worth noting that initial layers is usually on the smaller side, this further incentives the use of higher precision on initial layers.

Mixed-precision schemes should therefore prioritize higher precision (such as LPC8 or maybe even full 8-bit precision) for early layers while utilizing NC8 for deeper, more noise-resilient layers to maximize energy savings. The magnitude-based filtering showed some promise, where applying more precise multipliers to the top $\approx 10\%$ of weights by magnitude recovered significant accuracy, but generally this strategy needs some more improvement to be competitive to the layer-wise strategy. Also, this strategy is limited by its inability to exploit the lower latency of the NC multiplier at the hardware level, as the multiplier must be prepared to handle the most complex case in any given cycle.

One question raised in the introduction was to answer how powerful the ability to change precision for a finished system post implementation is, and if there is a meaningful difference between the lowest and highest precision setting that could allow for a more day-to-day operation adjustment scope. The conclusion here is that the accuracy difference between full HPC inference and full LPC inference is too small to make this a meaningful operation evaluation. Maybe if if using multiple different nets with the same structure but different weights. There are standard net architectures, like ResNet, VGG, or MobileNet, that could in theory be used this way, but this use-case seems quite far-fetched.

5.3 Designing Neural Networks for Multiple Precision

There do not seem to be any critical neural network design choices (like amount, size or types of layers) that should to be taken into consideration if planning to use approximate mixed-precision floating-point arithmetic. The results do not show any hard connections between any neural network traits and accuracy drop from precision loss. The best approach is to optimize the neural network to achieve the best accuracy per parameter, then apply mixed-precision afterwards. There seems to be a lesser accuracy loss, on larger nets, but this is probably because large nets are inherently more resilient because they have more redundant parameters.

5.4 Improvements and Further Research

The current implementation highlights several areas for optimization:

- **Accumulator Design:** The high cost of the floating-point accumulator is a primary bottleneck. Future work could explore more optimized structures, such as delayed normalization with a fixed-point (Kulisch) accumulator, to bridge the area and latency gap between floating-point and fixed-point MAC units.
- **Suitability for Large-Scale Models:** While this study focused on small and medium CNNs, literature and results on outlier handling suggest that floating-point formats are inherently better suited for the distribution shifts seen in Transformers and Large Language Models.
- **Approximation-Aware Training:** Integrating arithmetic approximations into the training loop could allow networks to learn to compensate for specific arithmetic errors, potentially allowing for even more aggressive approximations without accuracy loss.
- **System Complexity:** Implementing approximate floating-point multipliers on FPGAs is currently more labor-intensive than utilizing standard INT8 DSP slices. An important measure of usefulness of this approach depends on whether the reduction in re-quantization logic and the flexibility of dynamic precision can outweigh the custom implementation effort.

References

[1] Zhenhao Li, Zhaojun Lu, Wei Jia, Runze Yu, Haichun Zhang, Gefei Zhou, Zhenglin Liu, and Gang Qu. Efficient approximate floating-point multiplier with runtime reconfigurable frequency and precision. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 71(7):3533–3537, 2024.

[2] Jinhao Li, Jiaming Xu, Shan Huang, Yonghua Chen, Wen Li, Jun Liu, Yaoxiu Lian, Jiayi Pan, Li Ding, Hao Zhou, Yu Wang, and Guohao Dai. Large language model inference acceleration: A comprehensive hardware perspective, 2025.

[3] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021.

[5] Shangqian Gao, Feihu Huang, Weidong Cai, and Heng Huang. Network pruning via performance maximization. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 9266–9276, 2021.

[6] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Deepika Bablani, Jeffrey L. Mckinstry, Steven K. Esser, Rathinakumar Appuswamy, and Dharmendra S. Modha. Efficient and effective methods for mixed precision neural network quantization for faster, energy-efficient inference, 2024.

[8] Mart van Baalen, Andrey Kuzmin, Suparna S Nair, Yuwei Ren, Eric Mahurin, Chirag Patel, Sundar Subramanian, Sanghyuk Lee, Markus Nagel, Joseph Soriaga, and Tijmen Blankevoort. Fp8 versus int8 for efficient deep learning inference, 2023.

[9] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[10] Alex Krizhevsky. Cifar dataset, 2009. Accessed: 12.01.2025.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[12] Eivind Weidemann. Fpga cnn image classifier, 2025. Accessed: 2026-02-20.

Appendix A
Code

```c
float approx_mul(float a, float b, int mode) {
    union { float f; uint32_t u; } ua = { a }, ub = { b }, ur;

    // Extract sign, exponent, mantissa
    uint32_t sign_a = ua.u >> 31;
    uint32_t sign_b = ub.u >> 31;
    uint32_t exp_a  = (ua.u >> 23) & 0xFF;
    uint32_t exp_b  = (ub.u >> 23) & 0xFF;
    uint8_t mant_a  = (ua.u >> 16) & 0x7F; // 7 bits
    uint8_t mant_b  = (ub.u >> 16) & 0x7F;

    uint8_t mant_a_u  = (ua.u >> 20) & 0x07; // upper 3 bits
    uint8_t mant_b_u  = (ub.u >> 20) & 0x07;

    uint8_t mant_a_l  = (ua.u >> 17) & 0x07; // next 3 bits
    uint8_t mant_b_l  = (ub.u >> 17) & 0x07;


    // Handle zero or denormal inputs quickly
    if (exp_a == 0 || exp_b == 0)
        return 0.0f;

    // Add exponents
    uint32_t exp_out = exp_a + exp_b - 127;

    // ADD mantissa (not mul)
    uint8_t mant_sum = mant_a + mant_b;


    uint32_t error_correction;

    if (mode == 0) {
        // No correction
        error_correction = 0;
    } else if (mode == 1) {
        // Low precision correction
        error_correction = (((mant_a_u * mant_b_u) << 1) + 1);
```

```
    } else {
        // High precision correction

        uint32_t res1 = mant_a_u * mant_b_u;
        uint32_t res2 = mant_a_u * mant_b_l;
        uint8_t res3 = mant_a_l * mant_b_u;
        uint8_t res4 = mant_a_l * mant_b_l;

        error_correction = ((((res1 << 3) + res2) << 3) + ((res3 << 3) + res4)) >> 5;
    }

    uint32_t mant_approx_prod;

    // Check for mantissa overflow
    if ((mant_sum + error_correction) >= 128) {
        if(error_correction == 0) {
            mant_approx_prod = mant_sum - 128;
            exp_out += 1;
        } else {
            mant_approx_prod = ((mant_sum + error_correction - 128) >> 1);
            exp_out += 1;
        }

    } else {
        mant_approx_prod = mant_sum + error_correction;
    }

    // Combine sign, exponent, mantissa
    uint32_t sign_out = sign_a ^ sign_b;
    mant_approx_prod &= 0x7F; // ensure 7 bits
    ur.u = (sign_out << 31) | (exp_out << 23) | (mant_approx_prod << 16);

    return ur.f;
}
```

Listing A.1: Approximate multiplication function

```
    class SmallCNN(nn.Module): # 34 058 parameters
    def __init__(self, num_classes=10):
        super(SmallCNN, self).__init__()

        # Convolutional Feature Extractor with BatchNorm
        self.features = nn.Sequential(
            # Block 1
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
```

```
            nn.MaxPool2d(kernel_size=2, stride=2), # Output:
    32 x 16 x 16

            # Block 2
            nn.Conv2d(in_channels=16, out_channels=32,
    kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # Output:
    64 x 8 x 8

            # Block 3
            nn.Conv2d(in_channels=32, out_channels=64,
    kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # Output:
    128 x 4 x 4
        )

        # Fully Connected Classifier
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=64*4*4, out_features=
    num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Listing A.2: Small and simple CNN

```
    class MedCNN(nn.Module): # 1 147 914 parameters
    def __init__(self, num_classes=10):
        super(MedCNN, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32,
    kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(in_channels=32, out_channels=64,
    kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),

```

```
            nn.Conv2d(in_channels=64, out_channels=128,
    kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=128 * 4 * 4, out_features
    =512),
            nn.ReLU(),
            nn.Linear(in_features=512, out_features=
    num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Listing A.3: Larger but still simple CNN

```
    class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_planes, planes, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_planes, planes, kernel_size
    =3,
                               stride=stride, padding=1, bias=
    False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3,
                               stride=1, padding=1, bias=False
    )
        self.bn2 = nn.BatchNorm2d(planes)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != self.expansion * planes
    :
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, self.expansion * planes,
    kernel_size=1,
                          stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * planes)
            )

    def forward(self, x):
        out = torch.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
```

```
        out = torch.relu(out)
        return out

class ResNet9(nn.Module): # 43 226 parameters
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet9, self).__init__()
        self.in_planes = 16

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.layer1 = self._make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 32, num_blocks[1], stride=2)
        self.linear = nn.Linear(32 * block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = torch.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = nn.functional.avg_pool2d(out, out.size()[3])
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

Listing A.4: ResNet9

```
    class ConvBNReLU(nn.Sequential):

    def __init__(self, in_planes, out_planes, kernel_size=3, stride=1, groups=1):
        padding = (kernel_size - 1) // 2
        super(ConvBNReLU, self).__init__(
            nn.Conv2d(in_planes, out_planes, kernel_size, stride, padding, groups=groups, bias=False),
            nn.BatchNorm2d(out_planes),
            nn.ReLU6(inplace=True)
        )

class InvertedResidual(nn.Module):
```

```
12
13      def __init__(self, inp, oup, stride, expand_ratio):
14          super(InvertedResidual, self).__init__()
15          self.stride = stride
16          hidden_dim = int(round(inp * expand_ratio))
17          self.use_res_connect = self.stride == 1 and inp == oup
18
19          layers = []
20          if expand_ratio != 1:
21              # pw (pointwise expansion)
22              layers.append(ConvBNReLU(inp, hidden_dim,
    kernel_size=1))
23
24          layers.extend([
25              # dw (depthwise convolution)
26              ConvBNReLU(hidden_dim, hidden_dim, stride=stride,
    groups=hidden_dim),
27              # pw-linear (pointwise projection) - Note: No ReLU
     here (Linear Bottleneck)
28              nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
29              nn.BatchNorm2d(oup),
30          ])
31          self.conv = nn.Sequential(*layers)
32
33      def forward(self, x):
34          if self.use_res_connect:
35              return x + self.conv(x)
36          else:
37              return self.conv(x)
38
39  class MobileNetV2(nn.Module): # ~ 2 200 000 parameters
40      def __init__(self, num_classes=1000, width_mult=1.0):
41          super(MobileNetV2, self).__init__()
42          block = InvertedResidual
43          input_channel = 32
44          last_channel = 1280
45
46          # Setting for the blocks: [t (expansion), c (
    out_channels), n (repeats), s (stride)]
47          interverted_residual_setting = [
48              [1, 16, 1, 1],
49              [6, 24, 2, 2],
50              [6, 32, 3, 2],
51              [6, 64, 4, 2],
52              [6, 96, 3, 1],
53              [6, 160, 3, 2],
54              [6, 320, 1, 1],
55          ]
56
57          # Building the first layer
58          input_channel = int(input_channel * width_mult)
```

```
59          self.last_channel = int(last_channel * width_mult) if
    width_mult > 1.0 else last_channel
60
61          # Initial Feature extraction
62          features = [ConvBNReLU(3, input_channel, stride=1)]
63
64          # Building inverted residual blocks
65          for t, c, n, s in interverted_residual_setting:
66              output_channel = int(c * width_mult)
67              for i in range(n):
68                  stride = s if i == 0 else 1
69                  features.append(block(input_channel,
    output_channel, stride, expand_ratio=t))
70                  input_channel = output_channel
71
72          # Building last several layers
73          features.append(ConvBNReLU(input_channel, self.
    last_channel, kernel_size=1))
74
75          self.features = nn.Sequential(*features)
76
77          # Classifier
78          self.classifier = nn.Sequential(
79              nn.Dropout(0.2),
80              nn.Linear(self.last_channel, num_classes),
81          )
82
83          # Weight initialization
84          self._initialize_weights()
85
86      def forward(self, x):
87          x = self.features(x)
88          # Global Average Pooling
89          x = x.mean([2, 3])
90          x = self.classifier(x)
91          return x
92
93      def _initialize_weights(self):
94          for m in self.modules():
95              if isinstance(m, nn.Conv2d):
96                  nn.init.kaiming_normal_(m.weight, mode='
    fan_out')
97                  if m.bias is not None:
98                      nn.init.zeros_(m.bias)
99              elif isinstance(m, nn.BatchNorm2d):
100                 nn.init.ones_(m.weight)
101                 nn.init.zeros_(m.bias)
102             elif isinstance(m, nn.Linear):
103                 nn.init.normal_(m.weight, 0, 0.01)
104                 nn.init.zeros_(m.bias)
```

Listing A.5: MobileNetV2