# Impact of using approximate FP multipliers in neural network 

A Degree Thesis<br>Submitted to the Faculty of the<br>Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona Universitat Politècnica de Catalunya<br>by<br>Leixin Jin

In partial fulfilment
of the requirements for the degree in
Telecommunications Technologies and Services ENGINEERING

Advisors: Francesc Moll, Josep Altet and Antonio Rubio

$$
\text { Barcelona, May } 2022
$$

## Abstract

In the last few years, approximate computing has been one of the most popular topics in fields like image recognition, image analysis, language processing, selfdriving, etc. Many scientists have been studying how to make use of approximate arithmetic units to improve the efficiency, reduce the power consumption and delays of neural networks implementation.

In this thesis, we proposed three approximate multipliers for the mantissas multiplication, the first one is designed to reduce the number of calculations by putting one segment of the result to ' 1 ' s . The second one is the Mitchell logarithmic multiplier and the third one is the logarithmic multiplier with a set-one adder to compensate for the negative error which is brought by the Mitchell multiplier.
In order to evaluate these three multipliers, we are going to use YOLOv3, based on the open-source neural network framework which is called Darknet. This framework is dedicated to doing object recognition of images and we obtain the results after each execution.

## Resum

En els últims anys, computació aproximada ha estat un dels temes més populars en camps com el reconeixement d'imatges, l'anàlisi d'imatges, el processament del llenguatge. Molts cientifics han estat estudiant com aprofitar l'ús d'unitats aritmètiques aproximades per millorar l'eficiència, reduir el consum d'energia i els retards en implementacions de xarxes neuronals.

En aquesta tesi proposem tres multiplicadors aproximats per la multiplicació de les mantisses. El primer està dissenyat per reduir el nombre de càlculs posant una part del resultat a un valor constant determinat. El segon és el multiplicador logarítmic de Mitchell i el tercer és el multiplicador logarítmic amb un carry per compensar l'error negatiu que provoca el multiplicador logarítmic.

Per a avaluar aquests tres multiplicadors, utilitzarem la xarxa neuronal YOLOv3, basada en el framework de xarxa neuronal de codi obert que s'anomena Darknet. Aquest framework està dedicat a fer reconeixement d'objectes d'imatges.

## Acknowledgments

First of all, I would like to thank my thesis advisors, Francesc Moll, Josep Altet, and Antonio Rubio for their help and guidance during the project.
Secondly, I would like to thank Jordi Fornt who helped me to solve the problem which get stuck in the thesis for almost 2 months.
Thirdly, I would like to thank Enric Morancho who shared the Yolov3 code to do the experimental part.

Finally, I would like to thank my family, friends, and classmates for their support during these academic years.

## Revision history and approval record

| Revision | Date | Purpose |
| :--- | :--- | :--- |
| 0 | $03 / 04 / 2022$ | Document creation |
| 1 | $15 / 05 / 2022$ | Document revision |
|  |  |  |
|  |  |  |
|  |  |  |

DOCUMENT DISTRIBUTION LIST

| Name | e-mail |
| :--- | :--- |
| Leixin Jin | leixin.jin@estudiantat.upc.edu |
|  |  |
| Francesc Moll | francesc.moll@upc.edu |
| Josep Altet | josep.altet@upc.edu |
| Antonio Rubio | antonio.rubio@upc.edu |


| Written by: |  | Reviewed and approved by: |  |
| :--- | :--- | :--- | :--- |
| Date | $03 / 04 / 2022$ | Date | $15 / 05 / 2022$ |
| Name | Leixin Jin | Name | Francesc Moll |
| Position | Project Author | Position | Project Supervisor |

## Table of contents

Abstract ..... 1
Resum ..... 2
Acknowledgments ..... 3
Revision history and approval record ..... 4
Table of contents ..... 5
List of Figures ..... 6
List of Tables: ..... 7

1. Introduction ..... 8
1.1. Motivation ..... 8
1.2. Goals ..... 8
1.3. Thesis outline ..... 8
1.4. Gannt diagram ..... 8
2. State of the art ..... 9
2.1. IEEE-754 ..... 9
2.2. Multiplication ..... 10
2.3. Related work ..... 11
3. Methodology / project development: ..... 12
3.1. Approximate multipliers ..... 12
3.1.1. $\quad$ Carry-in prediction multiplier ..... 12
3.1.2. Mitchell's multiplier ..... 13
3.1.3. Logarithmic multiplier with set-one adder ..... 15
3.1.4. Distribution of numbers ..... 17
3.2. Darknet ..... 17
3.2.1. YOLO ..... 17
3.2.2. Code implementation ..... 18
4. Results ..... 21
4.1. Simulation results ..... 21
4.2. Darknet result ..... 23
4.2.1. Comparison of different position of carry bit ..... 28
5. Budget ..... 31
6. Conclusions and future development: ..... 32
Bibliography: ..... 33
List of Figures
Figure 1.IEEE-754 Floating-point number format ..... 9
Figure 2.Floating-point multiplication diagram block ..... 11
Figure 3.Numeric example of carry-in prediction multiplier ..... 13
Figure 4.Numeric example 1 of Mitchell's multiplier. ..... 14
Figure 5.Numeric example 2 of Mitchell's multiplier. ..... 15
Figure 6.Numeric example of third multiplier ..... 16
Figure 7.Distribution of numbers ..... 17
Figure 8.Comparison between YOLO and other detectors. Figure extracted from[13] ..... 18
Figure 9.Example of YOLO ..... 18
Figure 10.Code implementation of multiplier 1 ..... 19
Figure 11.Code implementation of multiplier 2 ..... 19
Figure 12.Code implementation of multiplier 3 ..... 20
Figure 13.Trend line of multiplier 1 ..... 21
Figure 14.Trend line of multiplier 2. ..... 22
Figure 15.Trend line of multiplier 3. ..... 22
Figure 16.Histogram of relative error ..... 23
Figure 17.Percentage of detected objects ..... 23
Figure 18.Darknet example 1 ..... 24
Figure 19.Darknet example 2 ..... 24
Figure 20.Percentage of different detected objects of multiplier exact ..... 25
Figure 21.Percentage of different detected objects of multiplier 1 ..... 25
Figure 22.Percentage of different detected objects of multiplier 2 ..... 26
Figure 23.Percentage of different detected objects of multiplier 3 ..... 26
Figure 24.Perfect detection ..... 27
Figure 25.Darknet example 3 ..... 27
Figure 26. Darknet example 4 ..... 28
Figure 27.Numeric example of different position of the carry bit ..... 28
Figure 28.Percentage of detected objects for different position ..... 29
Figure 29.Darknet example 5 ..... 29
Figure 30.Darknet example 6 ..... 30

## List of Tables:

Table 1. Different types of standard IEEE-754 ..... 10
Table 2. Budget of the project ..... 31

## 1．Introduction

## 1．1．Motivation

Nowadays，a wide variety of applications of neural networks are using floating point numbers for arithmetic operations such as multiplication，since virtually all modern processors have embedded floating point units．However，the floating point standard IEEE－ 754 requires high power consumption and incur in significant delay，while numerous neural networks applications are inherently error－tolerant for computations．Therefore，in the case of designing specific processors，，implementing the approximate floating－point multiplier for these applications reduces power consumption and improves energy efficiency and time．

The context of this project is the DRAC project（Designing RISC－V－based Accelerators for next generation Computers）［14］，in which the UPC participates，that has as its goal the design of specific processors for different applications，among them，neural networks for image identification in automotive applications．In this context，it is worth exploring efficient ways to do the required calculations exploiting approximate arithmetic．

## 1．2．Goals

This work will contribute the evaluation of the impact which generated by using three approximate multiplier on a neural network to identify objects in images．This thesis aims to study three different approximate floating point multipliers in terms of accuracy．

## 1．3．Thesis outline

In the first chapter，a brief introduction，motivation，and goals of the thesis．In Chapter 2 it is reviewed the current state of the art in the field of approximate operation．

Chapter 3 describes the methodology that is followed to develop the approximate multiplier． And their results are represented in Chapter 4.

Chapter 5 describes the budget of the thesis．
Finally，the last chapter contains the conclusion of the results obtained and future development that could be contributed to the project．

## 1．4．Gannt diagram

| WPID | INTERNAL TASKID | Start Date | End Date | $\begin{aligned} & \text { N } \\ & \text { O } \\ & \text { Nָ } \\ & \text { O} \end{aligned}$ | $\begin{aligned} & \text { O} \\ & \text { O} \\ & \text { İ } \\ & \text { 무 } \end{aligned}$ |  |  |  |  |  | स ت N N |  | $\begin{aligned} & \infty \\ & \underset{\sim}{\sim} \\ & \underset{\sim}{\sim} \\ & \text { ָn } \end{aligned}$ | $\begin{aligned} & \text { O} \\ & \underset{\sim}{\lambda} \\ & \underset{\sim}{\tilde{N}} \end{aligned}$ | N त̈ त्N | $\begin{aligned} & \underset{\sim}{\underset{\sim}{7}} \\ & \underset{\sim}{\tilde{\sim}} \end{aligned}$ | N స̈ त्̃ | $\begin{aligned} & \text { N} \\ & \text { ì } \\ & \text { Ñ } \\ & \text { O} \end{aligned}$ | $\begin{aligned} & \text { ơ } \\ & \text { í } \\ & \text { N } \\ & \text { O} \end{aligned}$ | N N N N | $\begin{aligned} & \text { N } \\ & \underset{N}{N} \\ & \underset{\sim}{N} \end{aligned}$ | $\begin{aligned} & \text { O} \\ & \underset{\sim}{1} \\ & \text { Ñ } \\ & \text { O} \end{aligned}$ |  |  |  | N N N N |  | $\begin{aligned} & \text { N } \\ & \text { N్} \\ & \text { Ñ } \\ & \text { 认ָ } \end{aligned}$ | O N N N |  | $\begin{aligned} & \text { O} \\ & \text { N } \\ & \text { N} \\ & \text { Nָ } \end{aligned}$ | $\begin{aligned} & \underset{N}{N} \\ & \underset{\sim}{N} \\ & \underset{\sim}{\sim} \end{aligned}$ | $\begin{aligned} & \text { M } \\ & \text { I } \\ & \text { N} \\ & \text { Ni} \end{aligned}$ | $\begin{aligned} & \text { O} \\ & \text { さ̦ } \\ & \text { ÑN } \end{aligned}$ |  |  | $\begin{aligned} & \text { O} \\ & \text { d } \\ & \text { N} \\ & \text { O} \end{aligned}$ | $\begin{aligned} & \text { o } \\ & \text { N } \\ & \text { N} \\ & \text { 人̀ } \end{aligned}$ |  |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| WP1 | T1．1 | 2021－09－26 | 2021－10－07 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | T1．2 | 2021－10－07 | 2021－10－15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | T1．3 | 2021－10－15 | 2021－10－24 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| WP2 | T2．1 | 20211031 | 20211114 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | T2．2 | 2021－10－31 | 2021－11－14 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| WP3 | T3．1 | 2021－11－21 | 2021－12－05 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | T3．2 | 2021－12－05 | 2022－03－20 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| WP4 | T4．1 | 2021－09－26 | 2021－10－03 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | T4．2 | 2021－11－24 | 2021－11－31 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | T4．3 | 2022－04－10 | 2022－05－15 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## 2. State of the art

### 2.1. IEEE-754

The IEEE-754 is a standard for floating-point numbers which was published in 1985 by the Institute of Electrical and Electronics Engineers(IEEE).
This standard represents the floating-point number in three components: sign, exponent, and mantissa.
The sign represents the sign of the number. 0 represents a positive floating-point number and 1 represents a negative number.

The exponent represents positive or negative numbers for the exponent of the floatingpoint number with a base of 2 or 10 .
To calculate this part, it has to sum all the bits by natural binary and subtracts the bias which is $2^{n-1}-1$. The mathematical representation is

$$
\begin{equation*}
\text { - } 2^{n-1}+2^{n-2}+. .+2^{0}-\text { bias } \tag{1}
\end{equation*}
$$

For example, if we considering a floating point number of 16 bits with a base of 2 and the exponent width will be 5 bits. His bias will be 15 or ' 01111 ' in binary, and his range will be between -14('00001') to 15 ('11110').
The mantissa represents the significand or fractional part of the floating-point number. However, we have to consider that mantissa has one bit implicit which is not stored o presented. The mathematical representation is

$$
\begin{equation*}
\text { - } 1+\left(2^{-1}+2^{-2}+. .+2^{n}\right) \tag{2}
\end{equation*}
$$

For example, if the mantissa is ' 0111100000 ', the real mantissa will be ' 101111000000 '. This red ' 1 ' is an implicit bit which has to be considered when doing the computation.
The following equation is the numeric representation of a floating point number.

$$
\begin{equation*}
\text { - } O_{1}=(-1)^{s_{1}} * 1 . \text { mantissa } a_{1} * 2^{e_{1}-\text { bias }} \tag{3}
\end{equation*}
$$

Figure 1 presents a format of floating point numbers and a numeric example.


Result: $1 * 2^{-1} * 1.6875=0,84375$
Figure 1.IEEE-754 Floating-point number format

According to the format, the standard IEEE-754 has defined several types of precision. The most popular are single-precision(32 bits) and double-precision(64 bits). Table 1 represents the different types of floating-point.

| Types | Size | Sign | Exponent | Mantissa |
| :---: | :---: | :---: | :---: | :---: |
| Half-precision | 16 bits | $\operatorname{Bit}[15]$ | $\operatorname{Bit}[14 \ldots 10]$ | $\operatorname{Bit}[9 \ldots 0]$ |
| Single-precision | 32 bits | $\operatorname{Bit}[31]$ | $\operatorname{Bit}[30 \ldots 23]$ | $\operatorname{Bit}[22 \ldots 0]$ |
| Double-precision | 64 bits | $\operatorname{Bit}[63]$ | $\operatorname{Bit}[62 \ldots 52]$ | $\operatorname{Bit}[51 \ldots 0]$ |
| Quadruple-precision | 128 bits | $\operatorname{Bit}[127]$ | $\operatorname{Bit}[126 \ldots 112]$ | $\operatorname{Bit}[111 \ldots 0]$ |

Table 1. Different types of standard IEEE-754

The standard also includes arithmetic formats which are:
-Signed zeros $( \pm 0),+0$ is ' 0000000000000000 ' and -0 is ' 1000000000000000 '.

- Signed infinite $( \pm \infty),+\infty$ is ' 0111110000000000 ' and $-\infty$ is ' 1111110000000000 '.
-Subnormal numbers. A non-zero number smaller than smallest-number ('0 000010000000000 ' in case of half-precision).
-NaN (not a numbers). A number which all bits of exponent are 1's. For example, ' $x 11111$ xxxxxxxxxx' is a NaN in case of half-precision.


### 2.2. Multiplication

The standard IEEE-754 defines the floating-point multiplication in 3 parts: calculation of sign bit, exponent and the product of mantissa. The sign bit is calculated by an XOR operation of two operands and the exponent part is calculated by the addition of two exponents and the subtraction of bias. The product of mantissa is calculated by the multiplication algorithms. After the operation, it is needed to normalise the mantissa and exponent. The normalisation is based on: if the first bit of mantissa is 1 , the exponent part will add 1 and the final mantissa part is the following bits of the first one. In the case of 0 , the exponent part and mantissa part will be the same.
To summarise, the multiplication operands are :

- $O_{1}=(-1)^{s_{1}} * 1$. mantiss $a_{1} * 2^{e_{1}-\text { bias }}$
- $O_{2}=(-1)^{s_{2}} * 1$. mantissa $a_{2} * 2^{e_{2}-\text { bias }}$

And the result is:

- $R=(-1)^{s_{1} \oplus s_{2}} *\left(1\right.$. mantissa $_{1} * 1$. mantissa $\left._{2}\right) * 2^{e_{1+} e_{2}-\text { bias }}$

Figure 2 shows the diagram block of multiplication.


Figure 2.Floating-point multiplication diagram block

### 2.3. Related work

In the last decade, approximate computing was one of the popular fields that many researchers were exploring. The works in [1] and [2] show how Karatsuba Algorithm affects to single-precision floating-point multiplier in terms of power consumption and delays. The work in [3] proposed a Vedic multiplier which improved $21,7 \%$ of delays compared to a traditional multiplier. Moreover, the work in [4] presents a comparison between three different multipliers(Booth, Karatsuba and Vedic).
On the other hand, the logarithmic multiplier(Mitchell's multiplier) [5] is another solution for approximate computing. This paper demonstrates the maximum possible multiplication error will be $11,1 \%$ and the division error will be 12,5\%. In addition, the works in [6],[7],[8] and [9] present different improved logarithmic multipliers based on Mitchell's multiplier to achieve better reduction in term of power and delays.

## 3. Methodology / project development:

In this chapter, we will present all relevant methods that were used in the development of the thesis. In section 3.1, we will be introducing three approximate multipliers which we have explored. Explaining each multiplier works with numeric examples and the distribution of numbers for simulation by using C and studying the errors generated by each multiplier. These approximate multipliers will be applied in the multiplication of mantissa, which means we will not modify the exponent part and sign part of the floatingpoint number. In section 3.2, we will be using these multipliers in YOLOv3 and study how the results are affected each multiplier in different images compared to the exact multiplier of mantissa.

### 3.1. Approximate multipliers

### 3.1.1. Carry-in prediction multiplier

This multiplier is inspired by this paper [10] which is consist to separate the multiplication into three parts, the high significant bits part (H), the low significant bits part (L) and midlow significant part (ML). This multiplier will do the accurate multiplication on the high significant bits part and low significant bits part. The inaccurate multiplication will be applied in the mid-low bits part. The approximation is based on putting the results by 1's and bringing a carry to the left column.
As the complexity of the partial product is increased by number of bits. Which means if the operand has more bits, the total number of partial product will be more. For example, the multiplication of two operand of 4 bits will have 4 partial product. And in case of 8 bits, the partial product will be 8.

So, the advantage of this multiplier is that we can reduce the computation complexity of the partial product. For instance, if we use a Wallace tree to compute the partial product, using this method will reduce number of computation stage, number of full-adders and improves the latency. In the paper [10], in case of a $8 \times 8$ partial product evaluation, author demonstrates that the number of computation stage has reduced by 1 unit, from 4 to 3 . Furthermore, the numbers of full-adders is from 15 to 9 .

Figure 3 is a numeric example of this multiplier. These partial products are separated in 3 parts: yellow columns are low significant bits part, blue columns are mid-low significant bits part and red columns are high significant bits part. The blue columns are computed approximately. This means their result has been forced to 1 's and brought a carry(C) to the left column. In this example, the exact result is $39 * 50=1950$ and the approximate result is 1982 which has a relative error $1,64 \%$.

UNIVERSITAT POLITĖCNICA
DE CATALUNYA
BARCELONATECH


Figure 3.Numeric example of carry-in prediction multiplier

### 3.1.2. Mitchell's multiplier

This multiplier is known as the logarithmic multiplier by Mitchell [5]. The main idea is to simplify the multiplication and division only using shifts and additions.

So, Mitchell defines 4 steps to compute the multiplication and division: leading one detection, binary-logarithm converter, addition or subtraction, and logarithm-binary converter.

Step 1-Leading one detection. Shift a number left until finds the first ' 1 ' bit and we note this counter as characteristic. As mantissa always starts with ' 1 ', this step can be simplified. Which means, the characteristic will be a number of bit of the mantissa. For example, in half-precision, the characteristic always is 10 or ' 1010 ' in binary.

Step2-Binary-logarithm converter. The bits which are after the first ' 1 ' bit will be maintained. These bits are also called the fractional part. In case of floating numbers, these bits are bits of the mantissa without the hidden ' 1 '. Finally, the combination of these two numbers will form a logarithm number.
Step3-Addition or subtraction. The case of multiplication, the sum will be applied. In contrast, the subtraction will be applied.
Step4-logarithm-binary converter. After step 3 , we will have 2 results, one is the characteristic and one is the fractional part. To convert to binary, in the first place we have to check if the result of fractional part is greater or equal than $2^{n}$. The case of affirmative, the characteristic will be incremented by 1 . In contrast, the characteristic will be the same. In the second place, we have to convert characteristic to decimal and assign one ' 1 ' to this position. For example, if the characteristic is ' 101 ' in binary, we have to put one ' 1 ' in the position 5 of the final result. Then, the fractional part will be concatenated after the characteristic and other positions will be ' 0 '.
Figure 4 is a numeric example without carry of this multiplier, we consider $k_{i}$ is the characteristic(red) and $x_{i}$ is the fractional part(green). In this case, both number have the same leading one bit in position 5 . So, their characteristics is ' 101 '. The fractional part is '00111' and '10010' respectively. Finally, the result of the characteristic is '1010' and we have to put ' 1 ' to position 10 and concatenate the fractional part.

Step 1) $k_{1}=101$ and $k_{2}=101$.

| 5 | 4 | 3 | $\mathbf{2}$ | 1 | 0 |
| ---: | ---: | ---: | ---: | ---: | ---: |
| $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ |
| $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{0}$ |

Step 2) $x_{1}=00111$ and $x_{2}=10010$.

| 5 | 4 | 3 | 2 | 1 | 0 |
| ---: | ---: | ---: | ---: | ---: | ---: |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 |

Form a logarithm number:

$$
\begin{aligned}
& k_{1}, x_{1}=10110111 \\
& k_{2}, x_{2}=10110010
\end{aligned}
$$

Step 3) $k, x=k_{1}+k_{2}, x_{1}+x_{2}$

$$
\begin{gathered}
k=k_{1}+k_{2}=101+101=1010 \\
x=x_{1}+x_{2}=00111+10010=11001
\end{gathered}
$$

## Step 4)

1. Check if the fractional part is greater or equal than $2^{n}$. In this case, $n=5$.

$$
\begin{aligned}
x= & 11001=25>\left(2^{5}=32\right) ? \\
& N o \rightarrow k=1010, x=11001
\end{aligned}
$$

2. Convert characteristic to decimal.

$$
k=1010=10
$$

3. Assign one ' 1 ' to the position $k$ and concatenated the $x$.

| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Figure 4.Numeric example 1 of Mitchell's multiplier.

Figure 5 is a numeric example with a carry. We can see that the fractional part is bigger than $2^{5}$ or ' 100000 ' and it has to bring a carry to the characteristic. So, we have to increment the characteristic by 1 and put ' 1 ' to the position 11(1011) and concatenate the fractional part exclude the first bit.

Step 1) $k_{1}=101$ and $k_{2}=101$.

| $\mathbf{5}$ | $\mathbf{4}$ | $\mathbf{3}$ | $\mathbf{2}$ | $\mathbf{1}$ | $\mathbf{0}$ |
| ---: | ---: | ---: | ---: | ---: | ---: |
| $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ |
| $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{0}$ |

Step 2) $x_{1}=10111$ and $x_{2}=10010$.

| 5 | 4 | 3 | 2 | 1 | 0 |
| ---: | ---: | ---: | ---: | ---: | ---: |
| $\mathbf{1}$ | 1 | 0 | 1 | 1 | 1 |
| $\mathbf{1}$ | 1 | 0 | 0 | 1 | 0 |

Form a logarithm number:

$$
\begin{aligned}
& k_{1}, x_{1}=10110111 \\
& k_{2}, x_{2}=10110010
\end{aligned}
$$

Step 3) $k, x=k_{1}+k_{2}, x_{1}+x_{2}$

$$
\begin{gathered}
k=k_{1}+k_{2}=101+101=1010 \\
x=x_{1}+x_{2}=00111+10010=101001
\end{gathered}
$$

Step 4)

1. Check if the fractional part is greater or equal than $2^{n}$. In this case, $\mathrm{n}=5$.

$$
\begin{gathered}
x=101001=41>\left(2^{5}=32\right) ? \\
Y e s \rightarrow k=1010+1=1011, x=01001
\end{gathered}
$$

2. Convert characteristic to decimal.

$$
k=1011=11
$$

3. Assign one ' 1 ' to the position k and concatenated the x .

| 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 5.Numeric example 2 of Mitchell's multiplier.

### 3.1.3. Logarithmic multiplier with set-one adder

As Mitchell's multiplier always underestimates the results which is important in neuronal network application. Because this application will do a million or more multiplication to obtain a result. This accumulated negative errors will affect the decision made by the application. So, for this reason, the authors of the work [8] designed three method to correct this inherent error, but in this project we will use only the method set-one adder (SOA). This multiplier is based on Mitchell's multiplier, the main idea is computing the high significant bit part exactly and forcing the lower significant bit to ' 1 '. It also bring a carry bit in a midhigh significant part to correct the negative error. The position of carry bit is flexible, we can put this bit where we prefer. But in this project we have chosen the position 6 and the reason is explained in the section 4.2.1.

An example of this multiplier is given in Figure 6. In this example, we consider six bits(yyyyyyy) will be computed inexactly by forcing to 1 . And the carry bit is located in position 6 or ' 001000000 ' in binary. So, the approximate result is 491008 and the exact result is 499200 . This means the relative error is $1,64 \%$.

Step 1) $k_{1}=1001$ and $k_{2}=1001$.

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 1 | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ |
| 1 | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ |

Step 2) $x_{1}=001$ yyyyyy and $x_{2}=100$ yyyyyy .

| $\mathbf{9}$ | $\mathbf{8}$ | $\mathbf{7}$ | 6 | $\mathbf{5}$ | 4 | 3 | $\mathbf{2}$ | $\mathbf{1}$ | $\mathbf{0}$ |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $\mathbf{1}$ | 0 | 0 | 1 | $\mathbf{1}$ | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ |
| $\mathbf{1}$ | 1 | 0 | 0 | $\mathbf{1}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ |

Form a logarithm number:

$$
\begin{aligned}
& k_{1}, x_{1}=1001001 \text { yyyyyy } \\
& k_{2}, x_{2}=1001100 \text { yyyyyy }
\end{aligned}
$$

Step 3) $k, x=k_{1}+k_{2}, x_{1}+x_{2}+$ Carry.

$$
\begin{gathered}
k=k_{1}+k_{2}=1001+1001=10010 \\
x=x_{1}+x_{2}+\text { Carry }=001 \text { yyyyyy }+100 \text { yyyyyy }+001000000=110 \text { yyyyyy } \\
y=1 \rightarrow x=110111111
\end{gathered}
$$

Step 4)

1. Check if the fractional part is greater or equal than $2^{n}$. In this case, $\mathrm{n}=9$.

$$
\begin{aligned}
x= & 110111111=447>\left(2^{9}=512\right) ? \\
& N o \rightarrow k=10010, x=110111111
\end{aligned}
$$

2. Convert characteristic to decimal.

$$
k=10010=18
$$

3. Assign one ' 1 ' to the position $k$ and concatenated the $x$.

| 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 6.Numeric example of third multiplier

### 3.1.4. Distribution of numbers

As mentioned in section 3 each multiplier will be simulated in one million of multiplications. So, we had created random numbers between 0 and 1023 to do this simulation. Figure 7 shows the distribution of these numbers and we can observe that it is a uniform distribution.


Figure 7.Distribution of numbers

### 3.2. Darknet

Darknet [11] is an open source neural network framework written in C and CUDA by Joseph Redmon. It supports CPU and GPU computation and the source code is shared on the author's Github [12].
This framework has implemented many applications, such as Object detection, ImageNet classification, Text generation, and GAN Nightmare. But in this project, we will focus on Object detection to observe the behaviour of these multipliers in image recognition.

### 3.2.1. YOLO

"You only look once" (YOLO) [13] is a real-time object detection system implemented by Darknet. To do the detection, this network divides the images into regions and in each region, predicts the bounding boxes and probabilities. If the probability is greater than 0,5 , the network will print the object and save the image with detected objects. This model has advantages over other detectors in terms of accuracy improvement and time saving. Figure 8 which is taken from work [13] presents a comparison between YOLO and other detectors. We can observe that YOLO is extremely faster than the RetinaNET and has a similar result in mAP(mean average precision).


Figure 8.Comparison between YOLO and other detectors. Figure extracted from[13]
In addition, Figure 9 is an example of YOLO that the objects detected has been marked by bounding boxes. In this case, YOLO detects 3 objects and they are dog, bicycle and truck.


Figure 9.Example of YOLO

### 3.2.2. Code implementation

To obtain the results of each multiplier, we changed the code of the exact multiplier to each one of the three approximate multipliers considered in this project. The source code could be download on GitHub[12]. And we only replace the line where the mantissa multiplications are performed. The line which we have changed is named uint_32 uint32_mul(uint_32_t x, uint_32_t y) and is located in the function uint_16 half_mul(uint_16_t x, uint_16_t y) of class half.c[15].

Figure 10 shows the code implementation of the first approximate multipliers. As explained in the section 3.1.1, we will separate the partial products in 3 parts. The first 'for' is used to compute all the partial product. The second 'for' is used to calculate the exact result in low significant bits part. And the third 'for' is for the exact result in high significant bits part. Finally, we have to force the mid-low significant part to ' 1 ' and sum the carry, where we do it using OR gate with ' 2016 '. To sum the carry, we just using the operand ' + ' with '2048'.

```
static inline uint32_t _uint32_mul(uint32_t a, uint32_t b)
{
    //mult 1, posant 1's a les posicions mig-baix pes
    uint32_t productesParcials[11];
    int i;
    for (i = 0; i<11; i++){
        productesParcials[i] = ((1<<i) & b) * a;
}
    uint32_t sumaPartLowSignificand = 0;
    for (i = 0; i < 5 ; ++i) {
        // 31 = 11111, els 5 bits més baix
        sumaPartLowSignificand = sumaPartLowSignificand + (productesParcials[i] & 31);
}
    sumaPartLowSignificand = sumaPartLowSignificand & 31;
    uint32_t sumaPartHighSignificand = 0;
        for (i = 1; i < 11; ++i) {
            //2096128 = 111111111110000000000, agafa els 11 bits de més pes
            sumaPartHighSignificand = sumaPartHighSignificand + (productesParcials[i] & 2096128);
    }
    //sumaPartHighSignificand = sumaPartHighSignificand & 2096128;
    //2048 es el carry a l'esquerra de la columna critica
    // 2016 = 11111100000
    uint32_t result = sumaPartHighSignificand | 2016 |sumaPartLowSignificand + 2048;
    //printf("%d\n", result);
    return (result);
```

Figure 10.Code implementation of multiplier 1
Figure 11 shows the code implementation of the second approximate multipliers. As we mentioned in the section 3.1.2, the step 1 can be simplified because of the mantissa's feature. So, we only need to do a sum of fractional part and check if this result is greater or equal than $2^{10}$ to see if the normalisation is needed.

```
static inline uint32_t _uint32_mul(uint32_t a, uint32_t b)
{
    uint32_t partA = a & 1023;
    uint32_t partB = b & 1023;
    uint32_t suma = partA + partB;
    if(suma >= 1024){
        return suma << 11;
    }else{
        return (suma << 10) | (1<<20);
    }
}
```

Figure 11.Code implementation of multiplier 2

Figure 12 shows the code implementation of the third approximate multipliers. The procedure is the same as the Mitchell multiplier. The unique change is when doing the sum, we will sum only 4 first bits(AND gate with 960). Furthermore, the carry and the bits which are forced to ' 1 ' is done by ' +127 ' or ' 1111111 ' in binary.

```
static inline uint32_t _uint32_mul(uint32_t a, uint32_t b)
{
    uint32_t partA = a & 1023;
    uint32_t partB = b & 1023;
    //cas de posar bit a la posició 7, 960->896 , 127->255
    //cas de posar bit a la posició 5, 960->992, 127->63
    //aquests valors es per la posició 6
    uint32_t suma = (partA & 960) + (partB & 960) +127;
    if(suma >= 1024){
        return suma << 11;
    }else{
        return (suma << 10) | (1<<20);
    }
}
```

Figure 12.Code implementation of multiplier 3

## 4. Results

In this chapter, we present the results obtained from applying the methodology of Chapter 3. This chapter is divided into two sections, the first is the simulation of one million multiplications in C for the three multipliers considered. We will display the trend-line between exact result and approximate result, and evaluate the errors introduced by each multiplier. In the second section we will evaluate the results obtained from applying an inexact multiplier in Darknet. In this section, we consider 25 images with different fields to observe the effect of each multiplier in a real application.

### 4.1. Simulation results

In this section, we will present a trend line of three multipliers on Figure 13, 14 and 15. We can observe that the first multiplier is introducing a little error because his trend line is extremely straight shown in Figure13. In the second place, Mitchell multiplier has introduced more errors, but these errors are negatives. That means this multiplier tends to underestimate the result, we can see this characteristic on Figure 14. In the Figure 14, we can observe that all points are under the orange diagonal. Finally, the last multiplier, due to a set-one adder, some approximate results are greater than the real results. The points are situated in the left-hand of the orange diagonal show this effect. In addition, this multiplier also limited the total number of results, because we forced 6 low significant bits to ' 1 ' and the result will be a combination of 4 bits. So, in Figure 15 we can see the trend line behaves like a ladder(horizontal lines).


Figure 13. Trend line of multiplier 1.


Figure 14. Trend line of multiplier 2.


Figure 15.Trend line of multiplier 3.

In addition, we will use the relative error to evaluate the errors of multipliers considered. The Figure 16 is a histogram of relative error, the first observation is that the error of Carry-in prediction multiplier is very small, these errors are always under than $1 \%$. The second observation is that the errors of Mitchell multiplier are negatives, this characteristic proves this multiplier always underestimates the result. The third
observation is that the third multiplier has reduced the error. In this multiplier, most of errors are under of $8 \%$ compare to the $11 \%$ of Mitchell multiplier.


Figure 16. Histogram of relative error.

### 4.2. Darknet result

In this section, we will explain the results obtained by Darknet in a set of 25 images. The main idea is to make a comparison between the exact multiplier and approximate multipliers in general aspects and special cases.
Figure 17 presents a percentage of detected objects (199 objects in total) of each multiplier. There are two interesting points, the first point is that the exact floating point of 16 bits is better than the exact floating point of 32 bits. The images which demonstrate this point is shown in Figure 18. In this case, we can observe that the multiplier of 16 bits detected one more house in the first image and one more teddy bear in the second image. The second point is that the carry-in prediction multiplier has detected more objects than the exact of 16 bits in this set of images.


Figure 17.Percentage of detected objects


Figure 18.Darknet example 1
In addition, Mitchell's multiplier has a very bad result, this multiplier only detected 49,7\% of objects. In contrast, the third multiplier due to the set-one adder has improved a 18,1\% of accuracy. We can observe this effect on Figure 19 which contains some example of this improvement.


Figure 19.Darknet example 2

On the other hand, the total number of objects that can be identified in the 25 images are 30 objects which are: person, bicycle, car, bus, bird, horse, cow, backpack, etc. From all of them, we have chosen 9 objects that have appeared more than 1 time in different pictures and they are person, horse, cow, bottle, cup, wine glass, orange, dinning table and teddy bear. Figure 20, 21, 22 and 23 present a percentage of these objects of different multiplier, we can observe that the carry-in prediction multiplier has detected the same or more objects than the exact of 16 bits. Considering this set of images, we can conclude that the carry-in prediction multiplier is the best of these four multipliers in general aspect, because it has detected more objects than others.

Percentage Of Detected Objects
Exacte (16)


Figure 20.Percentage of different detected objects of multiplier exact

Percentage of Different Objects Detected


Figure 21.Percentage of different detected objects of multiplier 1

## Percentage Of Detected Objects

$\log 1$


Figure 22.Percentage of different detected objects of multiplier 2

Percentage of Different Objects Detected log2


Figure 23.Percentage of different detected objects of multiplier 3

Another interesting study we have performed is to count how many images which all the objects have been perfectly detected. Figure 24 presents the number of times of perfect detection of each multiplier. The carry-in prediction multiplier continues to be the best detector in this case. This multiplier has 2 perfect detections more than the other multipliers.


Figure 24.Perfect detection

On the other hand, we will present some special cases that we found in the Darknet results. Figure 25 presents pictures that have many multiple equal item at the same time. In this situation, we can observe that Mitchell's multiplier(Log1) behaves very poorly. For example, in the picture of oranges, this multiplier only detected 3 oranges, and in the picture of teddy bears also is missing many teddy bears and chairs. So, we conclude that Mitchell's multiplier is inadequate for application that needs to distinguish the multiple equal item in the same image.


Figure 25.Darknet example 3

Figure 26 is an example where all the multipliers have detected incorrectly in the same object, 4 multipliers have detected the bag as a cake. So, as YOLO do the detection by separating the regions into bounding boxes and if the object border is quite confused like this situation, the application will detected incorrectly.


Figure 26.Darknet example 4

### 4.2.1. Comparison of different position of carry bit

As the work [8] had put the carry bit in a position of mid-high significant part, in this section we want to use the Darknet to find the best position for the carry bit. Considering the best position is the position which detects more objects possible and less incorrectly detection possible. So, we tried the carry bit in position 7, 6 and 5 and Figure 27 is a numeric example how to put the carry bit in different position. In the Figure 26, these blue bits correspond the carry bit and the position of carry bit also affect the number of bits of the fractional part will compute exactly. For example, if the carry bit is on position 7 , the fractional part (green bits) will only have 2 bits exactly and in case of position 6 , will have 3 bits exactly.

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $\mathbf{1}$ | $\mathbf{0}$ | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| $\mathbf{1}$ | $\mathbf{1}$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Case 7)

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Case 6)

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $\mathbf{1}$ | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| $\mathbf{1}$ | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Case 5)

| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- | :--- |
| $\mathbf{1}$ | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| $\mathbf{1}$ | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Figure 27.Numeric example of different position of the carry bit

In addition, to decide which position is the best, we are going to evaluate the percentage of total detected object. Figure 28 presents a percentage of these 3 positions. We found that the carry bit in position 6 has detected more objects than other positions.

Percentage of detected objects


Figure 28.Percentage of detected objects for different position
One interesting finding is that the position 7 detects less objects(15 objects) than other positions except the situation when image has multiple equal item at same time. Then, the position 7 is better than the other position. Figure 29 shows an example of this special case, in both pictures of position 7 has detected more objects than others positions. So, if the application needs to distinguish the similar things, the position 7 will be a good solution for this situation.


Figure 29.Darknet example 5

In addition, the reason that we do not choose the position 7 as the best position is demonstrated in the Figure 30. This position will generate many pictures with incorrect detection, 5 pictures of 25 . The other multiplier only generate 1 image with incorrect
detection. For example, in the picture 1), the fork is the wrong detection. In the pictures 2 ) and 4), Darknet has detected banana in each picture. And in the picture 3 ), there are 2 objects(dinning table and apple) are wrong detection. So, the position 7 increases the probability of incorrect detection and this is a big problem in the application like self-driving.


Figure 30.Darknet example 6

## 5. Budget

In this section, we are going to estimate the cost of the project. As mentioned in the introduction, this project will use a framework to determine the results. So, the implementation is only needs a computer with high computational capacity. In this case, we use our personal computer and get access to the remote server which is provided by UPC.

The main cost of the project is the salary of the researchers are involved in this work. This project has two roles, junior researcher and three senior researchers. The junior researcher will have an hourly wage of $10 € /$ hour and the senior researcher will have an hourly wage of $20 € /$ hours.

Considering this project was completed in 25 weeks. The table 2 summarises the budget of the project.

| Item | Amount | Cost | Dedication | Total cost |
| :---: | :---: | :---: | :---: | :---: |
| Computer | 1 | $800 €$ |  | $800 €$ |
| Junior researcher | 1 | $10 € /$ hour | $20 \mathrm{~h} /$ week | $5000 €$ |
| Senior researcher | 3 | $20 € / \mathrm{hour}$ | $2 \mathrm{~h} /$ week | $3000 €$ |
| Total |  |  |  |  |

Table 2. Budget of the project

## 6. Conclusions and future development:

The main goal of this project was to implement different approximate multipliers which are inspired by works [5], [8] and [10] to perform the multiplication more efficiently. To evaluate the result, we will apply these multipliers in a neural network application (YOLOv3) to do an object detection, we obtained interesting results about these multiplier utilisation.
The evaluation results of a set of 25 images in the section 4,2 demonstrate the following conclusions: the first is that the floating-point of 16 bits is better than the 32 bits in general aspect because the multiplier of 16 bits has detected more objects than 32 bits. The second conclusion is that if we force the low significand bit part to ' 1 ', the result is better than the exact. This conclusion is improved in the first multiplier and the third multiplier, the first multiplier has detected 3 more objects than the exact multiplier of 16 bits, and the third multiplier has increased $18,1 \%$ of accuracy compare to the Mitchell's multiplier.
In conclusion, these three multipliers reduced the complexity of computation. The first one reduced 6 columns of partial product. And the last two converted the multiplication to a single sum. In addition, the result of the first and the third multiplier was great because these two multipliers maintained a good level of accuracy, the former is even better than the exact; the latter reduced only $5,6 \%$ of accuracy.
As future work, there are different approaches that could be investigated. For example, implement this multiplier in a FPGA application in order to study the improvements in terms of area, power consumption, and delays. The second approach could be to simulate this multiplier in a particular field, i.e, self-driving.

## Bibliography:

[1] S. Arish and R. K. Sharma, "An efficient floating point multiplier design for high speed applications using Karatsuba algorithm and Urdhva-Tiryagbhyam algorithm," 2015 International Conference on Signal Processing and Communication (ICSC), Noida, 2015, pp. 303-308. doi: 10.1109/ICSPCom.2015.7150666
[2] Anand Mehta, C.B.Bidhul, Sajeevan Joseph, Jayakrishanan.P. "Implementation of Single Precision Floating Point Multiplier using Karatshu Algorithm". VIT University Vellore, Tamil Nadu, 632014, India, 2013.
[3] Paldurai.K, Dr.K.Hariharan, "FPGA Implementation of Delay Optimized Single Precision Floating point Multiplier",2015 International Conference on Advanced Computing and Communication Systems (ICACCS -2015), Jan. 05 - 07, 2015, Coimbatore, INDIA
[4] Ravi Kishore Kodali, Lakshmi Boppana and Sai Sourabh Yenamachintala, "FPGA Implementation of Vedic Floating point Multiplier", Department of Electronics and Communication Engineering National Institute of Technology,Warangal, 506004, INDIA
[5] Jhon N.Mitchell, "Computer Multiplication and Divisiong Using Binary Logarithms", pp.512-517.
[6] Min Soo Kim, Alberto A. Del Barrio, Leonardo Tavares Oliveira, Román Hermida, "Efficient Mitchell's Approximate Log Multiplier for Convolutional Neural Networks". IEEE TRANSACTIONS ON COMPUTERS, Vol.68, No.5, May 2019
[7] Uros Lotric, Ratko PiliPovic, Patricio Bulie, "A Hybrid Radix-4 and Approximate Logarithmic Multiplier for Energy Efficient Image Processing", Faculty of Computer and Information Science, University of Ljubljana, 1000 Ljubljana, Slovenia, 2021.
[8] Weiqiang Liu, Jiahua Xu, Danye Wang, Chenghua Wang, Paolo Montuschi, "Design and Evaluation of Approximate Logarithmic Multipliers for Low Power Error-Tolerant Applications. IEEE TRANSACTIONS ON CIRCUIT AND SYSTEMS-I REGULAR PAPERS, Vol.68, No.5, SEPTEMBER 2018
[9] Mohammad Saeed Ansari, Bruce.F.Cockburn, Jiehan, "An improved Logarithmic Multiplier for EnergyEfficient Neural Computing", IEEE TRANSACTIONS ON COMPUTERS, Vol.70, No.4, APRIL 2021
[10] Kartikeya Bhardwaj, Pravin S.Mane, Jorg Henkel, "Power and Area-Efficient Approximate Wallace Tree Multiplier for Error-Resilient Systems".
[11] Darknet web site. https://pireddie.com/darknet/
[12] Darknet source code. https://github.com/pjreddie/darknet
[13] YOLOv3 web site. https://pireddie.com/darknet/yolo/
[14] Drac web site. https://rdi.upc.edu/ca/ssri/projectes-institucionals/sectors-emergents-projectes/drac
[15] Mike Acton, 'Branch-free implementation of half-precision (16bits) flaoting point', 2006.
[16] Thesis architecture's code, GitHub. https://github.com/leixin-jin/TFG-Darknet

