UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

SEMINAR

Transprecision floating point

Dorian Brmbota

Mentor: *Izv. prof. dr. sc. Josip Knezović*

Zagreb, srpanj 2019.

CONTENTS

1. Introduction

With the progress of computer science new algorithms have emerged. Many of these algorithms rely on hardware with high FLOPS (floating-point operations per second). Increasing it becomes one of the most important optimization task today. On the other hand in low-power embedded platforms energy consumption is limiting factor. Transprecision floating-point is solution which covers both of those problems. Majority of this operations are done in *binary32* (single precision) and *binary64* (double precision) floating-point format. By reducing precision of floating-point to half or less we can vectorize operations and reduce number of needed operations in total. This would lead to reduced energy consumption and speed-up in algorithm execution. Experiments showed that execution time can be decreased by 12% and memory accesses can be reduced by 27% on average, leading to a reduction of energy consumption up to 30% [7]. The only problem that comes with reduced precision is do we still get acceptable algorithm results? I will analyze that in this work but for beginning let us see what the floating-point is according to IEEE-754 standard [6].

2. IEEE-754 floating-point standard

Floating-point arithmetic is a way of storing real numbers and calculating operations between them. Floating-point format is used to represent a finite subset of real number. Formats are characterized by their radix, precision, and exponent range, and each format can represent a unique set of floating-point data. These five are basics formats:

- Three binary formats, with encodings in lengths of 32, 64, and 128 bits.
- Two decimal formats, with encodings in lengths of 64 and 128 bits.

Other floating-point formats are derived from these basic formats.

2.1. Binary floating-point format

There are two types of radix defined by standard - decimal and binary. Decimal format, as the name suggests, uses 10 as base for exponent part (figure 2.1). Although it's more precise for representing decimal numbers, operations are much slower then in binary format. Because of that decimal radix isn't very interesting and from now on I'll be describing only binary format.

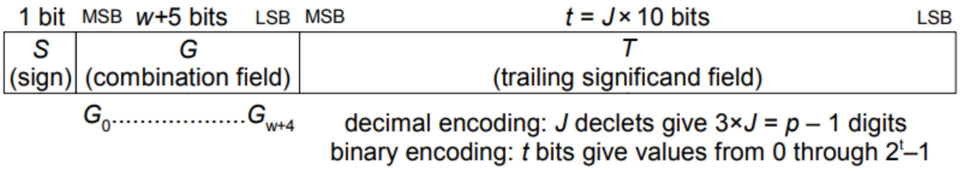

1 bit
MSB
w+5 bits
LSB
MSB
t = J × 10 bits
LSB
S
(sign)
G
(combination field)
T
(trailing significand field)
G0...................Gw+4
decimal encoding: J declets give 3×J = p – 1 digits
binary encoding: t bits give values from 0 through 2^t–1

Figure 2.1: Decimal format - trailing part can be encoded in binary or decimal [6]

Binary format is made out of three parts (figure 2.2):

- 1-bit sign *S*
- *w*-bit biased exponent $E = e + bias$
- $(t = p - 1)$-bit trailing significand field digit string $T = d1d2 \ldots dp - 1$; the leading bit of the significand, *d0*, is implicitly encoded in the biased exponent *E*.

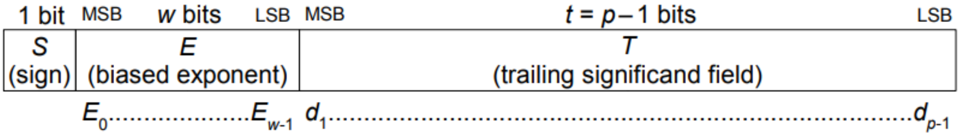
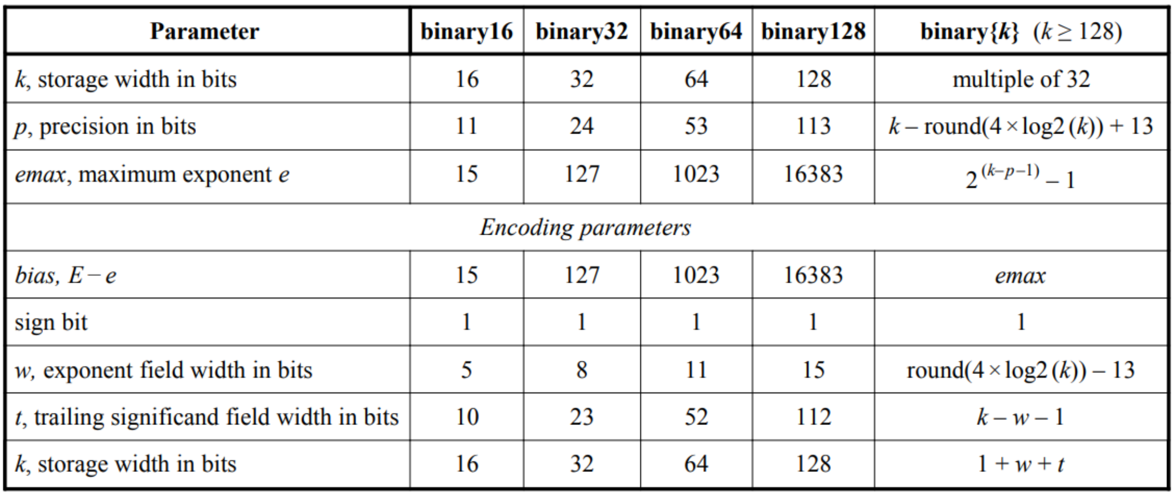


Figure 2.2: Binary format [6]

The 1-bit sign *S* part contains information if the number is positive (0) or negative (1). Range that format can cover is determined by exponent part. By variating *w* we change the smallest and largest number representable by floating-point. The trailing significand part is used for storing fractional part of number. Precision of floating-point depends on *t*. The more bits it has the more precise numbers it can store. Formula for calculating value from binary format:

- If $E = 2^w - 1$ and $T \neq 0$ then value is NaN regardless of *S*
- If $E = 2^w - 1$ and $T = 0$ then value is $(-1)^S * (+\infty)$
- If $1 \leq E \leq 2^w - 2$ the value of the corresponding floating-point number is $(-1)^S * 2^{E-bias} * (1 + 2^{1-p} * T)$
- If $E = 0$ and $T \neq 0$ then the value of the corresponding floating-point number is $(-1)^S * 2^{emin} * (0 + 2^{1-p} * T)$
- If $E = 0$ and $T = 0$ then the value is $(-1)^S * (+0)$

In figure 2.3 are formats defined by standard. I'll be using *binary32* and *binary16* for comparison with new derived formats.



Parameter	**binary16**	**binary32**	**binary64**	**binary128**	**binary{*k*}** ($k \geq 128$)
k, storage width in bits	16	32	64	128	multiple of 32
p, precision in bits	11	24	53	113	$k - \mathrm{round}(4 \times \log2(k)) + 13$
emax, maximum exponent *e*	15	127	1023	16383	$2^{(k-p-1)} - 1$
Encoding parameters					
bias, E − e	15	127	1023	16383	*emax*
sign bit	1	1	1	1	1
w, exponent field width in bits	5	8	11	15	$\mathrm{round}(4 \times \log2(k)) - 13$
t, trailing significand field width in bits	10	23	52	112	$k - w - 1$
k, storage width in bits	16	32	64	128	$1 + w + t$

Figure 2.3: Binary formats defined by standard [6]

3. Floating point arithmetic

IEEE-754 also defines some floating-point operations that have to be implemented in order to be compatible with standard. Also, before we implement operations we need to implement rounding attributes. Standard defines that we treat every number as infinitely precise and then round it to fit in the destination's format. There are four rounding modes required by IEEE-754 for binary format:

- *roundTiesToEven* - the floating-point number should be nearest to the infinitely precise result. Also if magnitude is at least $b^{emax}(b - 1/2 * b^{1-p})$ it should be rounded to ∞ with no change in sign. If two number are equally nearest to an unrepresentable infinitely precise result the one with an even least significant digit is chosen.
- *roundTowardPositive* - the result shall be the format's floating-point number (possibly $+\infty$) closest to and no less than the infinitely precise result.
- *roundTowardNegative* - the result shall be the format's floating-point number (possibly $-\infty$) closest to and no greater than the infinitely precise result.
- *roundTowardZero* - the result shall be the format's floating-point number closest to and no greater in magnitude than the infinitely precise result.

There is also *roundTiesToAway* rounding mode similar to *roundTiesToEven* but it's optional for binary format.

With rounding modes implemented we can go to defining required floating-point operations. Following operations should be implemented for all supported arithmetic formats [6].

1. **General operations**
   - *inputFormat* **roundToIntegralTiesToEven**(x) - rounds x to the nearest integral value, with halfway cases rounding to even.
   - *inputFormat* **roundToIntegralTowardZero**(x) - rounds x to an integral value toward zero.

- *inputFormat* **roundToIntegralTowardPositive**(x) - rounds x to an integral value toward positive infinity.
- *inputFormat* **roundToIntegralTowardNegative**(x) - rounds x to an integral value toward negative infinity.
- *inputFormat* **roundToIntegralTiesToAway**(x) - rounds x to the nearest integral value, with halfway cases rounding away from zero.
- *inputFormat* **roundToIntegralExact**(x) - rounds x to an integral value according to the applicable rounding-direction attribute.
- *inputFormat* **nextUp**(x) - is the least floating-point number in the format of x that compares greater than x.
- *inputFormat* **nextDown**(x) - is $-\text{nextUp}(-x)$.
- *inputFormat* **remainder**(x, y) - is defined for finite x and y regardless of the rounding-direction attribute by the mathematical relation $r = x - y * n$, where n is the integer nearest the exact number x/y.
- *inputFormat* **minNum**(x, y) - is x if $x < y$, y if $y < x$.
- *inputFormat* **maxNum**(x, y) - is y if $x < y$, x if $y < x$.
- *inputFormat* **minNumMag**(x, y) - is x if $|x| < |y|$, y if $|y| < |x|$.
- *inputFormat* **maxNumMag**(x, y) - is y if $|x| < |y|$, x if $|y| < |x|$.

2. ***logBFormat* operations** - for each supported arithmetic format, languages define an associated *logBFormat* to contain the integral values of $log_B(x)$. The *logBFormat* shall have enough range to include all integers between $\pm 2 * (emax + p)$ inclusive, which includes the scale factors for scaling between the finite numbers of largest and smallest magnitude. As we only observe binary formats B is equal to 2.

   - *inputFormat* **scaleB**(x, N) - is $x * b^N$ for integral values N. In case of binary format is $x * 2^N$.
   - *logBFormat* **logB**(x) - is floor($log_B(x)$). In case of binary format is floor($log_2(x)$)

3. **Artihmetic operations** - by standard implementations must provide the following *outputFormat* general-computational operations, for destinations of all supported arithmetic formats, and, for each destination format, for operands of all supported arithmetic formats with the same radix as the destination format.

   - *outputFormat* **addition**(x, y)

- *outputFormat* **subtraction**(x, y)
- *outputFormat* **multiplication**(x, y)
- *outputFormat* **division**(x, y)
- *outputFormat* **squareRoot**(x)
- *outputFormat* **fusedMultiplyAdd**(x, y, z) - is $(x * y) + z$.
- *outputFormat* **convertFromInt**(x)
- *intOutputFormat* **convertToIntegerTiesToEven**(x)
- *intOutputFormat* **convertToIntegerTowardZero**(x)
- *intOutputFormat* **convertToIntegerTowardPositive**(x)
- *intOutputFormat* **convertToIntegerTowardNegative**(x)
- *intOutputFormat* **convertToIntegerTiesToAway**(x)
- *intOutputFormat* **convertToIntegerExactTiesToEven**(x)
- *intOutputFormat* **convertToIntegerExactTowardZero**(x)
- *intOutputFormat* **convertToIntegerExactTowardPositive**(x)
- *intOutputFormat* **convertToIntegerExactTowardNegative**(x)
- *intOutputFormat* **convertToIntegerExactTiesToAway**(x)

4. **Conversion operations**
   - *outputFormat* **convertFromHexCharacter**$(hexCharacterSequence)$
   - *hexCharacterSequence* **convertFromHexCharacter**$(x, conversionSpecification)$

5. **Sign bit operations**
   - *inputFormat* **copy**(x) - copies a floating-point operand x to a destination in the same format, with no change to the sign bit.
   - *inputFormat* **negate**(x) - copies a floating-point operand x to a destination in the same format, reversing the sign bit.
   - *inputFormat* **abs**(x) - copies a floating-point operand x to a destination in the same format, setting the sign bit to 0 (positive).
   - *inputFormat* **copySign**(x, y) - copies a floating-point operand x to a destination in the same format as x, but with the sign bit of y.

6. **Comparisons** - there are two types of comparison operators - signaling and quiet. Differnece between them is that signaling raises exception if operation is invalid (in case of NaNs).
   - *boolean* **compareQuietEqual**(x, y)

- *boolean* **compareQuietNotEqual**(x, y)
- *boolean* **compareSignalingEqual**(x, y)
- *boolean* **compareSignalingGreater**(x, y)
- *boolean* **compareSignalingGreaterEqual**(x, y)
- *boolean* **compareSignalingLess**(x, y)
- *boolean* **compareSignalingLessEqual**(x, y)
- *boolean* **compareSignalingNotEqual**(x, y)
- *boolean* **compareSignalingNotGreater**(x, y)
- *boolean* **compareSignalingLessUnordered**(x, y)
- *boolean* **compareSignalingNotLess**(x, y)
- *boolean* **compareSignalingGreaterUnordered**(x, y)
- *boolean* **compareQuietGreater**(x, y)
- *boolean* **compareQuietGreaterEqual**(x, y)
- *boolean* **compareQuietLess**(x, y)
- *boolean* **compareQuietLessEqual**(x, y)
- *boolean* **compareQuietUnordered**(x, y)
- *boolean* **compareQuietNotGreater**(x, y)
- *boolean* **compareQuietLessUnordered**(x, y)
- *boolean* **compareQuietNotLess**(x, y)
- *boolean* **compareQuietGreaterUnordered**(x, y)
- *boolean* **compareQuietOrdered**(x, y)

7. **Conformance predicates**

   - *boolean* **is754version1985**($void$) - is true if and only if this programming environment conforms to IEEE 754-1985 standard.
   - *boolean* **is754version2008**($void$) - is true if and only if this programming environment conforms to IEEE 754-2008 standard.

8. **Other general operations**

   - *enum* **class**(x) - tells which of the following ten classes x falls into: signalingNaN, quietNaN, negativeInfinity, negativeNormal, negativeSubnormal, negativeZero, positiveZero, positiveSubnormal, positiveNormal, positiveInfinity
   - *boolean* **isSignMinus**(x) - is true if and only if x has negative sign.

- *boolean* **isNormal**(x) - is true if and only if x is normal (not zero, subnormal, infinite, or NaN).
- *boolean* **isFinite**(x) - is true if and only if x is zero, subnormal or normal (not infinite or NaN).
- *boolean* **isZero**(x) - is true if and only if x is ± 0.
- *boolean* **isSubnormal**(x) - is true if and only if x is subnormal.
- *boolean* **isInfinite**(x) - is true if and only if x is infinite.
- *boolean* **isNaN**(x) - is true if and only if x is a NaN.
- *boolean* **isSignaling**(x) - is true if and only if x is a signaling NaN.
- *boolean* **isCanonical**(x) - is true if and only if x is a finite number, infinity, or NaN that is canonical.
- *enum* **radix**(x) - is the radix b of the format of x, that is, two or ten.
- *boolean* **totalOrder**(x, y) - is true if $x < y$ with all other special cases also defined by standard.
- *boolean* **totalOrderMag**(x, y) - is totalOrder(abs(x), abs(y)).

Because I'll be only testing output precision (and not performance) depending on different floating-point formats I'll only make software implementation of rounding modes for new formats and use all other operations from already implemented formats (*binary32, binary16*).

4. Transprecision floating-point computing

Recently more and more new algorithms are made to work with approximate results rather then requiring high precision. With evolution of approximate-computing new paradigm has emerged - *transprecision computing*. Main concept of this paradigm is not to tolerate error but to deliver result inside required precision. Research done by this article [7] shows that 50% of energy falls of to floating-point operation in high computation applications. They also showed that 90% of operations can safely be scaled down to 8-bit and 16-bit formats so the results are still inside required precision. By doing that they could vectorize operations which saved them 30% of energy consumption and lead to decrease of execution time by 12%. This study also introduced two new types - *binary8* and *binary16alt* (figure 4.1). SW - *FlexFloat*[8] and HW - simple FPU (figure 4.2) implementation was made.

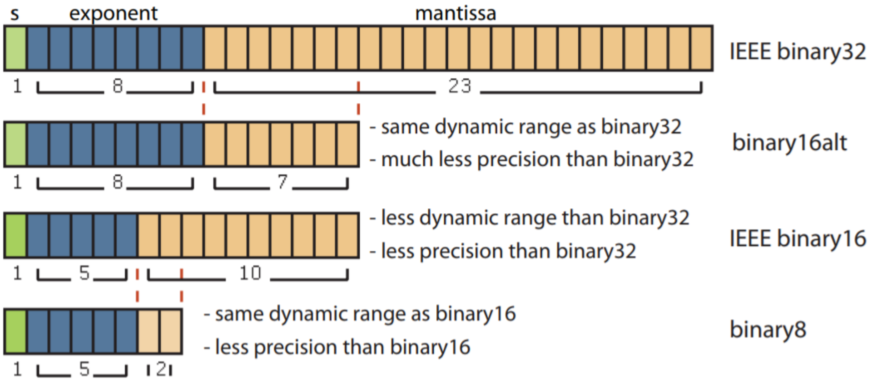

s
exponent
mantissa
IEEE binary32
1
8
23
- same dynamic range as binary32
- much less precision than binary32
binary16alt
1
8
7
- less dynamic range than binary32
- less precision than binary32
IEEE binary16
1
5
10
- same dynamic range as binary16
- less precision than binary16
binary8
1
5
2

Figure 4.1: Floating-point formats used in this work. [7]

Experimetns have been performed on six different algorithms:

- JACOBI applies the Jacobi method to a 2D heat grid
- KNN computes the k-nearest neighbors of an input value using euclidean distance

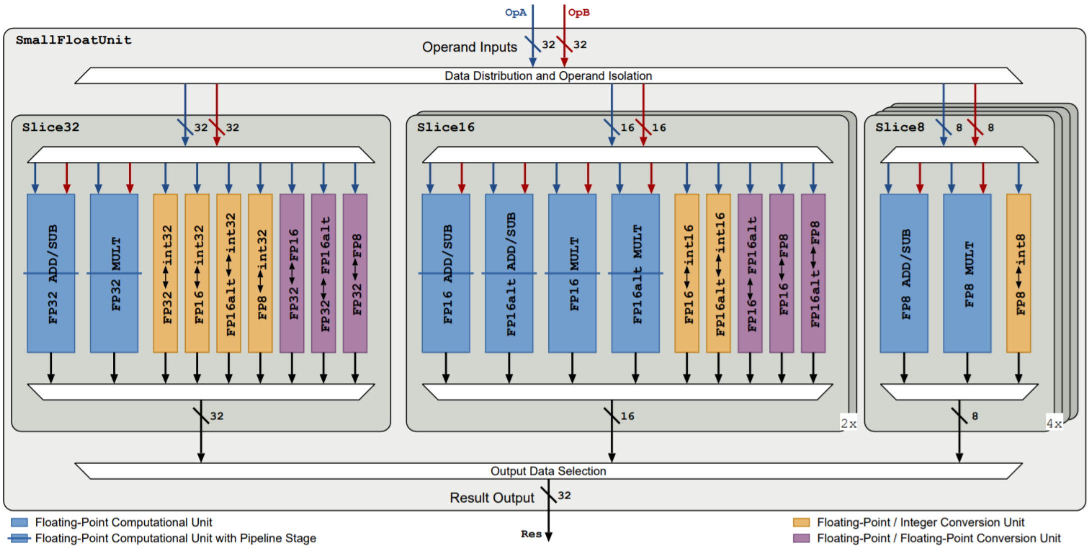
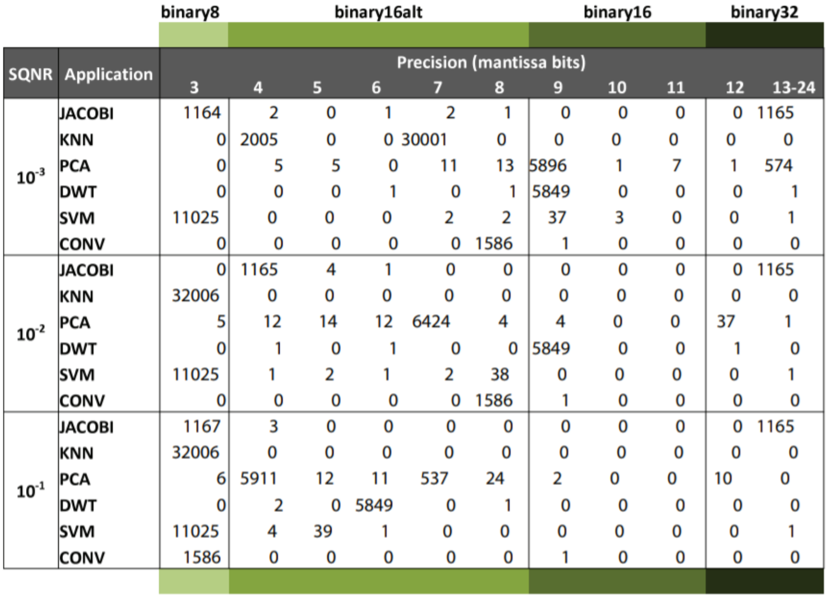


Figure 4.2: Simple floating-point unit. [7]

- PCA performs the principal component analysis
- DWT computes the discrete wavelet transform
- SVM is the prediction stage of a support vector machine
- CONV implements a 5×5 convolution kernel

Experiment I'll be looking at is precision tuning. It was mode for three precision requirements - $10^{-1}, 10^{-2}$ and 10^{-3}. This figure 4.3 shows results. Numbers in figure represents how many successful floating-point operations are scaled down to specific precision. The goal here was to use as smaller format as possible while maintaining result inside required precision. This experiment showed that many operations can be safely scaled to 8-bit and 16-bit formats. To extend this research I've created my own precision experiment on deep neural networks.



		binary8	binary16alt					binary16			binary32	
SQNR	Application	Precision (mantissa bits)										
		3	4	5	6	7	8	9	10	11	12	13-24
10^{-3}	JACOBI	1164	2	0	1	2	1	0	0	0	0	1165
	KNN	0	2005	0	0	30001	0	0	0	0	0	0
	PCA	0	5	5	0	11	13	5896	1	7	1	574
	DWT	0	0	0	1	0	1	5849	0	0	0	1
	SVM	11025	0	0	0	2	2	37	3	0	0	1
	CONV	0	0	0	0	0	1586	1	0	0	0	0
10^{-2}	JACOBI	0	1165	4	1	0	0	0	0	0	0	1165
	KNN	32006	0	0	0	0	0	0	0	0	0	0
	PCA	5	12	14	12	6424	4	4	0	0	37	1
	DWT	0	1	0	1	0	0	5849	0	0	1	0
	SVM	11025	1	2	1	2	38	0	0	0	0	1
	CONV	0	0	0	0	0	1586	1	0	0	0	0
10^{-1}	JACOBI	1167	3	0	0	0	0	0	0	0	0	1165
	KNN	32006	0	0	0	0	0	0	0	0	0	0
	PCA	6	5911	12	11	537	24	2	0	0	10	0
	DWT	0	2	0	5849	0	1	0	0	0	0	0
	SVM	11025	4	39	1	0	0	0	0	0	0	1
	CONV	1586	0	0	0	0	0	1	0	0	0	0

Figure 4.3: Precision tuning of program variables for three precision requirements. [7]

5. Floating-point precision comparison on DNNs

This experiment contains two projects - *tensor* and *neural_network*. Project *tensor*, similar to *numpy* in python, is library for working with matrices and tensors of arbitrary size. Type and format of tensor can be defined. If needed after every operation on tensor data it uses rounding for result to match given format. Implementation of rounding is taken from *FlexFloat* [8] and adjusted to work with *CUDA* [2]. Both *tensor* and *neural_network* are implemented in *CUDA* to be able to process big datasets. They also take advantage of *cuBLAS*[3], *cuDNN*[4] and *cuRAND*[5] libraries to achieve best performance. Project *neural_network* contains layers and activations needed for basic fully connected and convolutional neural networks.

Task for neural networks was to learn to recognize handwritten digits from 0-9. Dataset used for learning is *MNIST*[1]. *MNIST* contains 60000 labeled images with handwritten digits for training and 10000 for test. For training, before each epoch train set was randomly permuted and split into batches of size 60. After each epoch accuracy and loss was calculated from test set and saved for graph. Before I get to my first experiment I just want to define some expression that I will be using:

- fc($output\_size$) - fully connected layer
- relu() - Rectified linear unit - $relu(x) = max(x, 0)$
- softmax() - Cross entropy/log loss
- conv($kernel\_width \times kernel\_height \times \ output\_channels$)) - convolutional layer
- maxpool($kernel\_size, stride$) - Pooling layer using maximum
- SGD - Stochastic gradient descent

First experiment contained fully connected neural network with following configuration: $fc(784) \rightarrow relu() \rightarrow fc(100) \rightarrow relu() \rightarrow fc(10) \rightarrow softmax()$. Learning algorithm is SGD with momentum (parameters - learning rate $\epsilon = 0.01$ and momentum $\alpha = 0.5$). For referent case I set format to *binary32* (figure 5.1). It's also important to notice that my main goal wasn't to achieve best possible accuracy but to find formats with which I get results (loss) that are within 10^{-2} of referent case. Accuracy of neural network could be improved more with different deep learning techniques. With that in mind lets see how well 16-bit formats have performed (figure 5.2). Here I used *binary16* for weights and bias, but for gradients, to have more range because they can get small, I used *binary16alt*. Figures shows that *binary32* can be easily replaced with *binary16* and *binary16alt* without losing any accuracy at all. Now lets see if it's possible to scale down format even more. These are results for *binary8* (figure 5.3). Reason behind this results is too low learning rate. Multiplying learning rate with gradient results in value that is smaller then smallest representable by format and it is rounded to ± 0. To try ti avoid that I'll use Adam as learning algorithm with parameters, learning rate - $\epsilon = 0.01$, decay rates - $\rho_1 = 0.9$ and $\rho_2 = 0.999$ and small constant for numerical stabilization - $\delta = 10^{-8}$. The results (figure 5.4) show significant improvement that could be satisfying for some cases but we are still not inside 10^{-2} error which is our goal.

I'll try to utilize *binary8* format better in second experiment. Second experiment shows how convolutional neural networks works with different formats. Configuration of network: $conv(5 \times 5 \times 16) \rightarrow maxpool(3, 2) \rightarrow conv(5 \times 5 \times 32) \rightarrow maxpool(3, 2) \rightarrow fc(512) \rightarrow relu() \rightarrow fc(10) \rightarrow softmax()$. Learning algorithm is SGD with momentum (parameters - learning rate $\epsilon = 0.005$ and momentum $\alpha = 0.5$). Same as before first case is referent case in *binary32* format (figure 5.5). Second case with weights and bias in *binary16* format and grads in *binary16alt* format works equally good as in first experiment (figure 5.6). Now lets see how well *binary8* format performs in convolution. In first try (figure 5.7) we encounter same problem as in first experiment - learning rate is too low but now there are some improvements over epochs so lets try to set higher learning rate ($\epsilon = 0.05$). Results (figure 5.8) of this last case are inside requested error of 10^{-2} and *binary8* format is utilized successfully.

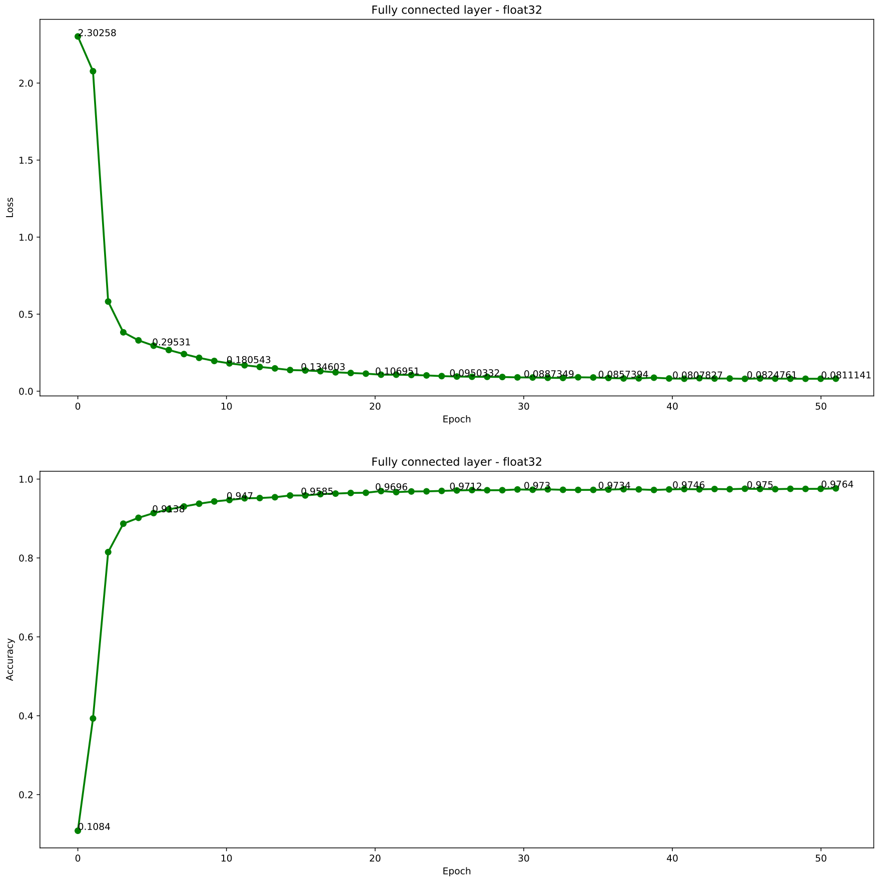


Figure 5.1: Loss and accuracy of fully connected network for *binary32* format.

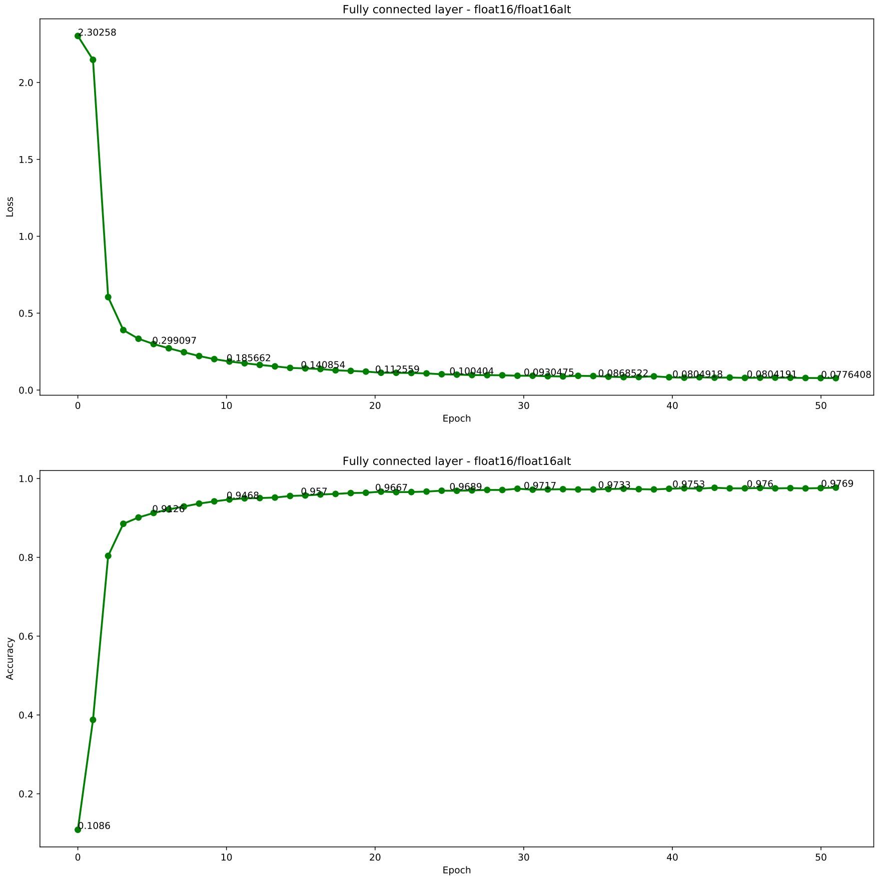


Figure 5.2: Loss and accuracy of fully connected network for *binary16* and *binary16alt* format.

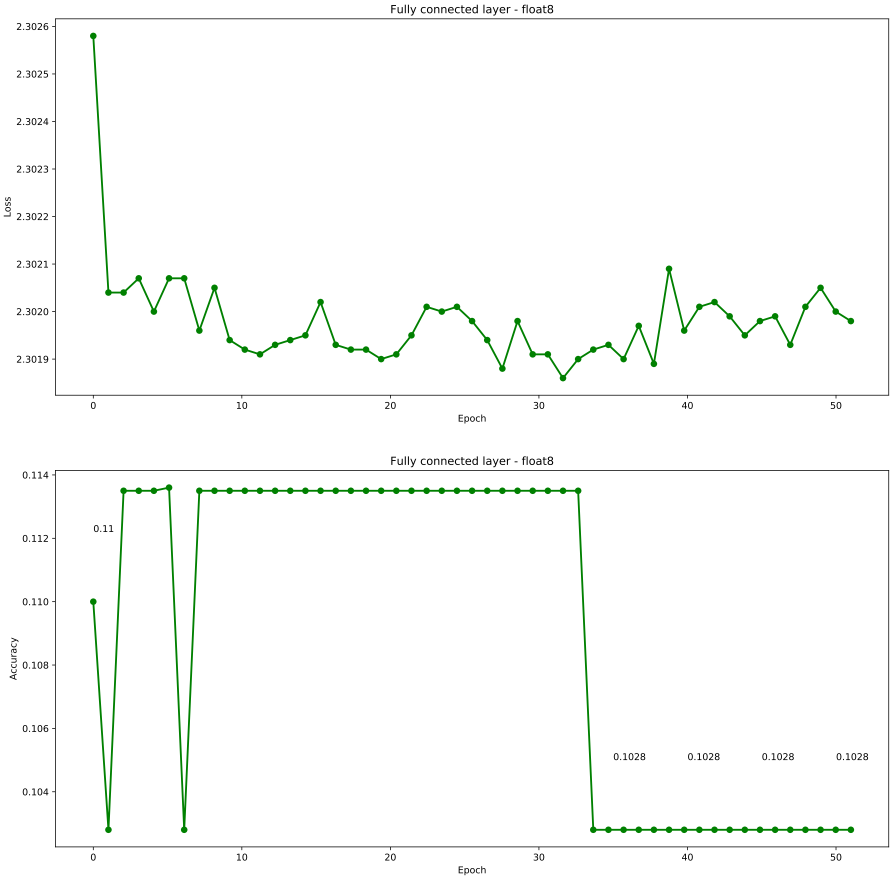


Figure 5.3: Loss and accuracy of fully connected network for *binary8* format.

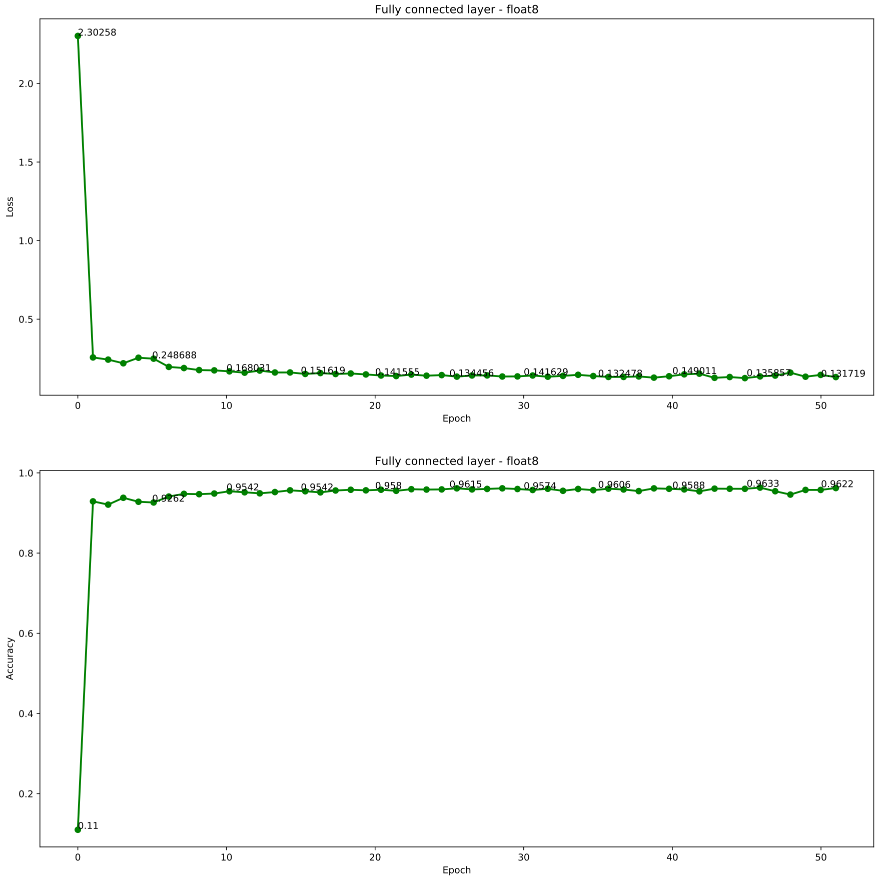


Figure 5.4: Loss and accuracy of fully connected network for *binary8* format with Adam.

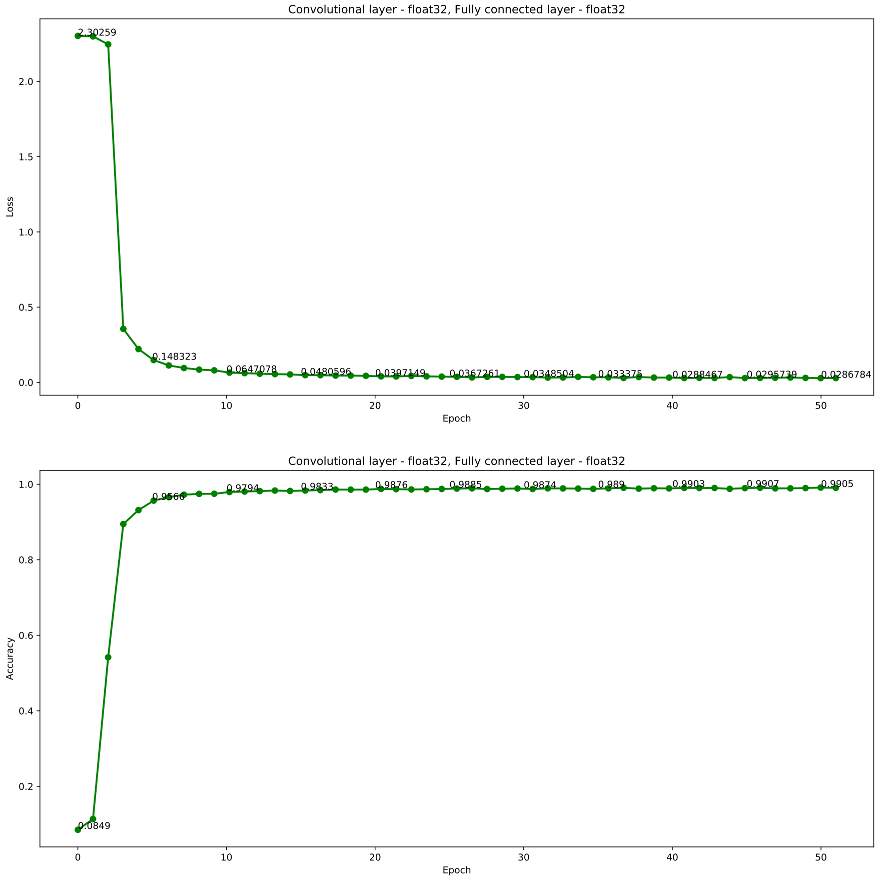


Figure 5.5: Loss and accuracy of convolutional neural network for *binary32* format.

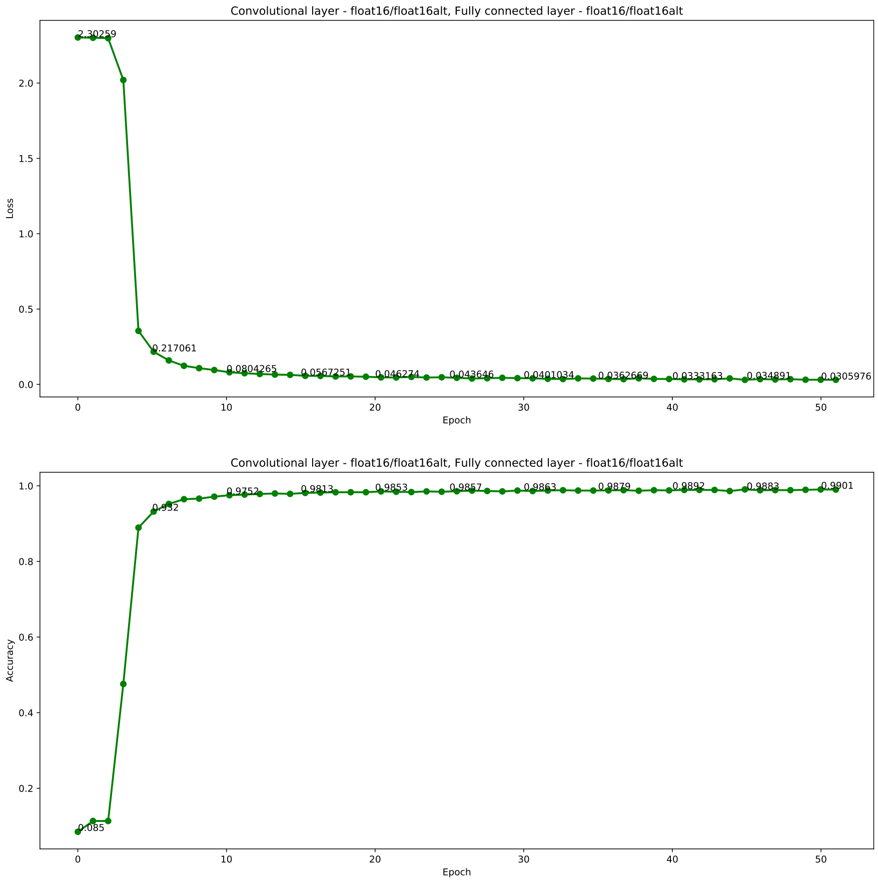


Figure 5.6: Loss and accuracy of convolutional neural network for *binary16* and *binary16alt* format.

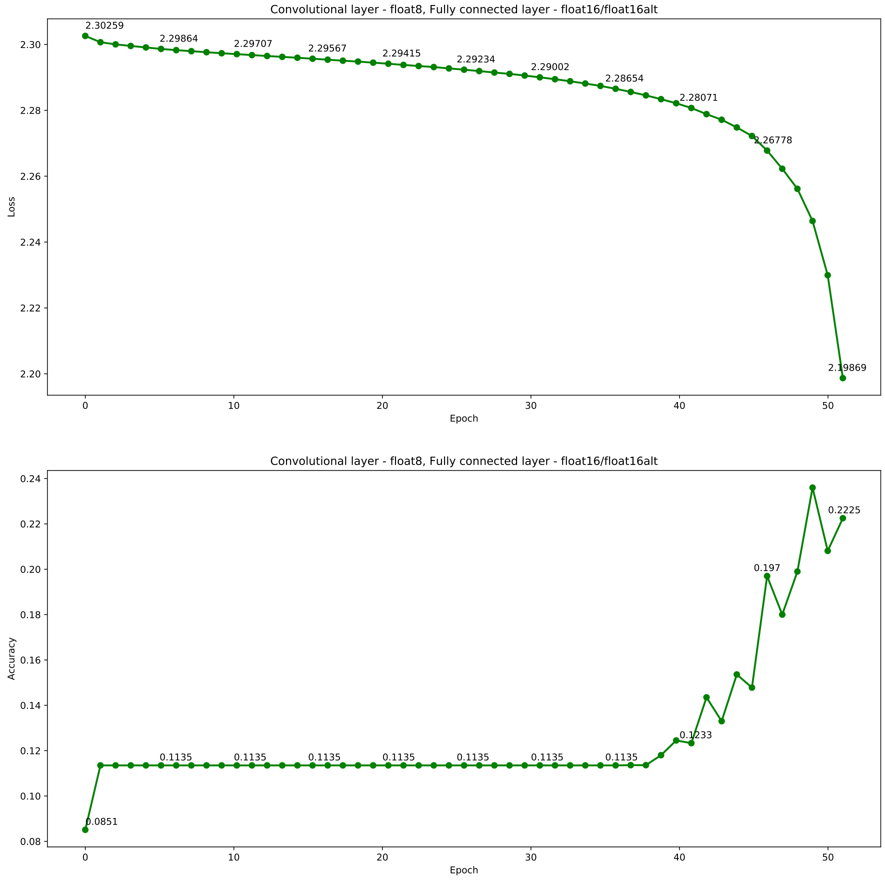


Figure 5.7: Loss and accuracy of convolutional neural network for *binary8* format.

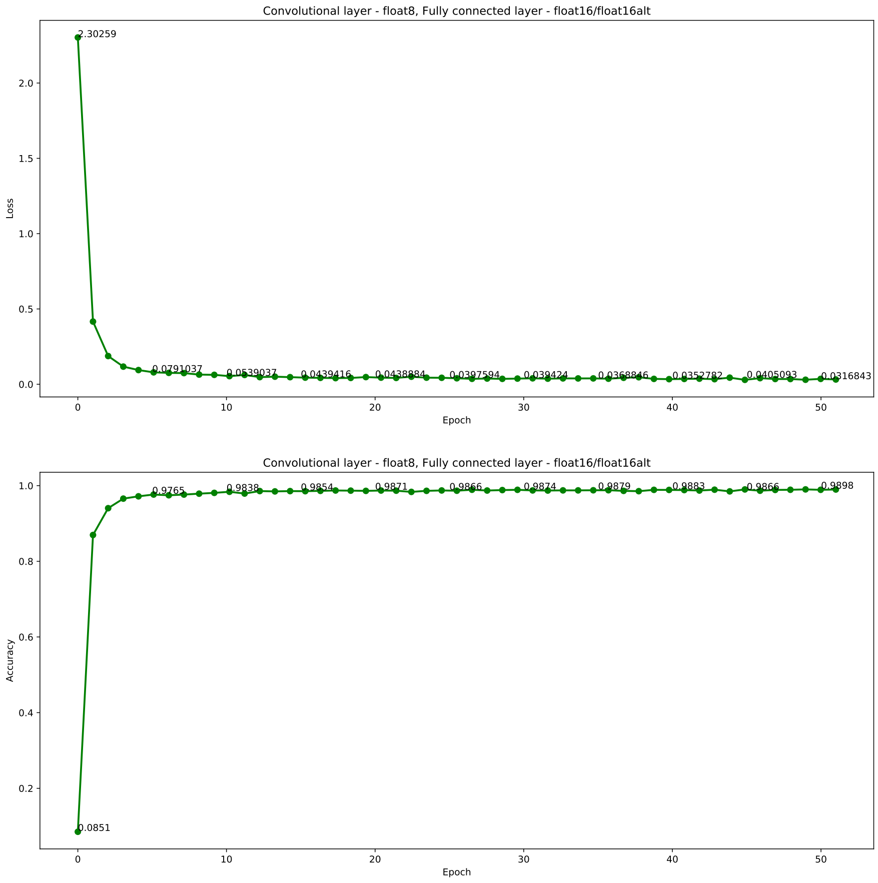


Figure 5.8: Loss and accuracy of convolutional neural network for *binary8* format with higher learning rate.

6. Conclusion

Experiments showed that it's possible to scale down operations to 8-bit and 16-bit and still get relevant results. Improvements are not only seen in energy savings and speed-up but also in memory consumption. In the last experiment there could be savings in memory over 2x on parameters and little less then 4x in gradients. This could be a huge difference in bigger datasets and bigger neural networks (measured in GB)! Transprecision computing is paradigm that definitely can be utilized in new algorithms. It can bring many advantages and improvements if implemented correctly. What remains next is implementation of floating-point unit, then it testing to see what are real energy savings and speed-up improvements.

7. Bibliography

[1] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`, 2019.

[2] NVIDIA. Cuda toolkit 10.1 update 1. `https://developer.nvidia.com/cuda-downloads`, 2019.

[3] NVIDIA. cublas 9.2. `https://developer.nvidia.com/cublas`, 2019.

[4] NVIDIA. cudnn 7.6. `https://developer.nvidia.com/cudnn`, 2019.

[5] NVIDIA. curand documentation. `https://docs.nvidia.com/cuda/curand/index.html`, 2019.

[6] IEEE Computer Society. Ieee standard 754 for floating-point arithmetic.

[7] Giuseppe Tagliavini, Stefan Mach, Davide Rossia, Andrea Marongiu, and Luca Benini. A transprecision floating-point platform for ultra-low power computing.

[8] Giuseppe Tagliavini, Stefan Mach, and Cristiano Malossi. Flexfloat. `https://github.com/oprecomp/flexfloat`, 2018.

8. Summary

Recently more and more new algorithms are made to work with approximate results rather then requiring high precision. With evolution of approximate-computing new paradigm has emerged - *transprecision computing*. Research shows that 50% of energy falls of to floating-point operation in high computation applications. They also showed that 90% of operations can safely be scaled down to 8-bit and 16-bit formats so the results are still inside required precision. By doing that they could vectorize operations which saved them 30% of energy consumption and lead to decrease of execution time by 12%. But to be able to alter floating-point some standard must be defined. IEEE 754-2008 is latest floating-point standard. It defines floating-point format and operations. After defining standard to test newly introduced floating-point formats experiments on neural network algorithms has been made. Result showed that *transprecision computing* can bring many improvements.