BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

DATA-DRIVEN DIGITAL CIRCUIT APPROXIMATION

DATY ŘÍZENÁ APROXIMACE ČÍSLICOVÝCH OBVODŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MATĚJ VÁLEK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Abstract

This Master's thesis deals with approximate implementations of arithmetic operations in image filters. In particular, it uses approximation techniques to adjust the multiplication operations in a non-trivial image filter. Several methods are employed, such as converting the floating-point multiplication to fixed-point multiplication, applying evolutionary algorithms, especially Cartesian genetic programming, to create new approximate multipliers that have an acceptable level of error, and at the same time, reduced filtering complexity. The result is a collection of approximate multipliers evolved with respect to the data distribution retrieved from the image filter. Approximate image filters that use evolved approximate multipliers are compared with the standard image filter on a set of images.

Abstrakt

Tato diplomová práce se zabývá aproximativní implementace aritmetických operací v obrazových filtrech. Zejména tedy využitím aproximativních technik pro úpravu způsobu násobení v netriviálním obrazovém filtru. K tomu je využito několik technik, jako použití převodu násobení s pohyblivou řadovou čárkou na násobení s pevnou řadovou čárkou, či využití evolučních algoritmů zejména kartézkého genetického programování pro vytvoření nových aproximovaných násobiček, které vykazují přijatelnou chybu, ale současně redukují výpočetní náročnost filtrace. Výsledkem jsou evolučně navržené aproximativní násobičky zohledňující distribuci dat v obrazovém filtru a jejich nasazení v obrazovém filtru a porovnání původního filtru s aproximovaným fitrem na sadě barevných obrázků.

Keywords

approximate computing, evolutionary algorithm, Cartesian genetic programming, fixed-point arithmetics, non-local denoising filter, approximate multipliers.

Klíčová slova

aproximované výpočty, evoluční algoritmy, Kartezské genetické programování, operace v pevné řadové čárce, ne-lokální odšumovací filtr, aproximované násobičky.

Reference

VÁLEK, Matěj. *Data-Driven Digital Circuit Approximation.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Data-Driven Digital Circuit Approximation

Declaration

Herebly I declare, that this project was prepared as original author work under the supervision of Prof. Lukáš Sekanina. Additional information were provided by Bc. Jan Klhůfek, and Ing. Vojtěch Mrázek ,Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Matěj Válek
May 13, 2021

Acknowledgements

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project „e-Infrastructure CZ – LM2018140“. Also I want to thank Bc. Jan Klhůfek, and Ing. Vojtěch Mrázek ,Ph.D. for helping me with conversion of multiplier representation from Verilog to CGP and from CGP to C. Big gratitude belongs to my parents for supporting me in the times of my studies.

Contents

1 Introduction **3**

2 Approximate computing **5**
2.1 Motivation for applying approximate computing 5
2.1.1 Natural need for approximations 5
2.1.2 Approximation in the floating-point arithmetic 6
2.1.3 Optimisation of efficiency 6
2.1.4 Approximation in hardware and software 6
2.2 Quality metrics 6

3 Evolutionary algorithms **8**
3.1 Principle of evolutionary algorithms 8
3.1.1 Optimization in context of evolutionary algorithm 8
3.2 Problem encoding 9
3.3 Mutation 9
3.4 Crossover 10
3.5 Selection mechanisms 11
3.6 EA in practice 11

4 Genetic programming **13**
4.1 Cartesian genetic programming 13
4.1.1 Method of CGP 13
4.1.2 Search method 15
4.2 Properties of CGP 15
4.3 Optimization of CGP 16

5 Non-local means denoising filter **17**
5.1 Basic idea of non-local denoising 17
5.2 Non-local means filter 18
5.3 Pixelwise implementation 18
5.4 Patchwise Implementation 19
5.5 Software implementation 19
5.6 Approximation computing in image processing 20
5.7 Approximation approaches 21
5.8 Approximation of a NL-means filter 21

6 Floating point to fix point arithmetic **23**
6.1 Fixed point representation 24

6.2 Fixed point arithmetic operations 24
6.3 Experiments with bit-width and fixed point representation 26
6.4 Casting operands to double bit width 28
6.5 Shifting operations in fixed-point arithmetic 31

7 EvoApproxLib **34**
7.1 What is EvoApproxLib 34
7.2 Experiments with 16 bit approximate multiplier 34

8 Approximation of a multiplier **38**
8.1 Specification of the approximation 38
8.2 Determining bit width for a new multiplier 39
8.3 Approximation method 39
8.3.1 Conversion of the circuit to CGP chromosome 39
8.3.2 Error computation based on simulation 40
8.4 Error metrics definition 40
8.5 Design of the approximation process 41
8.6 Using parallel simulation for better performance. 42

9 Acceleration of CGP with a new mutation operator **44**
9.1 Principles of the probabilistic map heuristic 44
9.1.1 Basic Principle of Back-propagation 45
9.2 Algorithm for creating the probabilistic map 46

10 Results of the approximation **48**
10.1 Approximation and program arguments 48
10.2 Results for 20-bit multiplier using the probability map mutation 50
10.2.1 Examining the evolution process 50
10.3 Results for 20-bit multiplier using classic mutation 53
10.4 Applying approximated multipliers with image filter to image dataset . . . 56
10.4.1 Resource Savings 58
10.5 Real life image denoised by approximated 20-bit multiplier 59

11 Conclusion **60**

Bibliography **61**

A Appendix **63**
A.1 CD content 63
A.2 Additional real life images denoised by approximated 20-bit multiplier . . . 64

Chapter 1

Introduction

In today's world, inquiry for computational power and energy-efficient systems is rapidly increasing. We all have mobile phones, laptops and other similar devices. On a bigger scale we are building more data centers and computing systems in general. Energy efficiency has become the major concern in the design of embedded systems. Energy consumed by our systems is increasing exponentially [6]. That is why modern system design is taking in account saving as much energy as possible. In this thesis we will discuss the methods of approximate computing (AC) to reduce power consumption of computer systems. Approximate computing is an engineering paradigm in which power consumption is traded for quality of the result. It is motivated by the fact that some applications are highly error-resilient and can thus be safely approximated.

An open problem is how to obtain high-quality approximate implementations of hardware and software. In recent years, evolutionary algorithms (EA) were presented as a useful approach in this direction. In particular, Cartesian genetic programming provided high quality approximated implementations of important circuit components [9]. This approach can be adopted in today's key applications such as deep learning or image processing because they are error resilient. Two types of approaches to approximate computing were developed: the first method assumes nothing about the distribution of the input data while the second method exploits particular data distributions to deliver even better results in approximate implementations [10].

The objective of my Master thesis is to develop an EA based approximation which exploits particular data distributions into an approximable image filter. The reason why we used image filtering as a case study in this thesis is that image processing is one of the best candidate fields in which approximate computing can be adopted. Images are very error resilient subjects due to the limitation of human sight. Thus there is great opportunity for this kind of approximations. In this thesis a non-trivial filter is the use case. When filtering an image, the result has a great potential for introducing a bigger error because the human eye and brain simply cannot distinguish the difference between the exact image and approximated one.

The rest of the thesis is organized as follows. In the chapter 2 basic theoretical principles of approximate computing are presented. Specifically we show how it can be done, why it is so popular at this time and how some mathematics background is utilized in approximate computing, mainly statistical analysis. Also as a part of the chapter 5, specific approaches for image filter approximations are presented.

The chapter 3 is devoted to theoretical background of evolutionary algorithms. In particular problem encoding, genetic operators and search algorithm are briefly introduced.

The special version of evolutionary algorithms, the so called Cartesian genetic programming is discussed in chapter 4. This technique is later used for the approximation process. In particular, for the design of complex approximate multipliers.

In chapter 6 the use case of this thesis, which is a non-local denoising filter is presented. This chapter is a milestone between theoretical and practical part of this thesis because in chapter 7 first basic technique for approximate computing is described, namely the transformation of floating-point to fixed-point number representation for arithmetic circuits.

In chapters 7 there are presented results for using already known n-bit approximate multipliers borrowed from EvoApproxLib [11]. Chapters 8 and 9 describe modified evolutionary techniques designed specifically for obtaining the best possible approximate multipliers for the chosen use case of this study. Moreover, a new learning technique is proposed in these chapters to construct a better evolutionary mutation operator leading to a faster evolutionary design process.

in Chapter 10, statistical information about the evolutionary design process is presented and resulting multipliers are validated on various color images.

Chapter 2

Approximate computing

In the modern world the inquiry for improving our systems in terms of energy saving and memory space has significantly risen. Usage of computing devices is growing exponentially and it is more evident that it will overgrow our current resource limits (transistors in chips etc.). For example, studies show, that information processed in next decade will grow 50 times more than today but the numbers of processors will increase only ten times [10].

Approximate computing may be a solution to this problem. The basic idea is that as we can relax quality requirements in some applications we could be able, thanks to some selective approximations or small changes to the specification, to make a huge difference in energy usage. So in some applications, where we can afford to have a little bit bigger error, we can trade-off this for higher efficiency in power usage. For example, performing k-means clustering algorithm, up to 50× energy saving can be achieved by allowing a classification accuracy loss of 5 percent [3].

We must say that AC could not be optimal every time in terms trading energy or power for quality of result (QoR). It is a very difficult task to accomplish. In general, it is important to choose right code/data segments for approximations. Simple uniform approximations could end up in non-acceptable quality loss. It is possible that if some heavy approximations are made, catastrophic errors may occur, such as segmentation faults [10].

2.1 Motivation for applying approximate computing

We shall now discuss some scenarios and opportunities where AC can be effective or even unavoidable [10].

2.1.1 Natural need for approximations

In some cases, approximation is inevitably needed. It happens in such cases where the input is noisy, small precision of input data is given, hardware is chosen badly or an error occurs in hardware, surprising additional load, or hard real-time constraints. For example, if we consider floating point operations where rounding is a frequent operation then approximations are inevitably needed [10].

2.1.2 Approximation in the floating-point arithmetic

In today applications (e.g. in image processing), almost all calculations are being made with practical zero or negligible error. And it has almost no effect on quality of the result (QoR). Problem is human vision is perceptually limited. Experiments shown that 98% of floating point operation and 91% of data accesses are approximable. Since logically lower bits in arithmetic operations have much lower impact on QoR than values around the MSB bits a approximation potential arises [10].

2.1.3 Optimisation of efficiency

In this report we will be looking into approximation of an image filter. Experiments shown that peak signal-to-noise ratio (PSNR) greater or equal to 30dB, and error value less than 10% is acceptable. If we use this fact, we are theoretically able by using AC improve energy efficiency, areas on the chip and performance significantly. As a example apart from image processing if we look at a hardware modification, it shows that if we can correctly reduce the refresh rate in DRAM, the precision of the functionality of these components remains almost the same, but memory access and storage of data will require significantly less energy [10].

2.1.4 Approximation in hardware and software

The goal when approximating a system is to find its sub-parts that are error-resilient and suitable for approximation. A common approach is Random/guided modification to the original implementation and statistical evaluation of the impact on the quality of result. Usual subsystems used for approximation in HW and SW: [1]

Software	Hardware
precision of number representation	bit width reduction
data storage strategies	intentional disconnecting of components
code simplification	timing changes
relaxed synchronization	power supply voltage changes
unfinished loops	fault injection

Table 2.1: Example of approximable subsystems in software and hardware.

2.2 Quality metrics

If we want to effectively balance an outcome, in other words to have an optimal ratio between error and power/area efficiency, we have to apply AC on different levels of the system - application software, processor, accelerator, memory etc. We use metrics summarized in table 2.2 as a common way for measuring the error which is introduced by AC. This metrics can be classified as application-specific or general-purpose [10].

[1]Source: http://www.fit.vutbr.cz/~sekanina/publ/ahs18/ahs18_tut_approx_comp_1.pdf

Quality metric(s)	Corresponding applications/kernels
Relative difference/ error from standard output	Monte Carlo, sparse matrix multiplication, Jacobi, discrete Fourier transform, MapReduce programs
PSNR and SSIM	MPEG, JPEG, rayshade, image resizer, image smoothing, OpenGL games
Pixel difference	Bodytrack (PARSEC), eon (SPEC2000), raytracer (Splash2), particle lter (Rodinia), volume rendering
Energy conservation across scenes	Physics-based simulation (e.g., collision detection, constraint solving)
Classication/clustering accuracy	k-nearest neighbor, k-means clustering, generalized learning vector quantization
Correct/incorrect decisions	nding Julia set fractals, jMonkeyEngine
Ratio of error of initial and nal gues	3D variable coefcient Helmholtz equation, image compression
Ranking accuracy	Bing search, supervised semantic indexing

Table 2.2: Examples of quality metrics used in approximate computing and typical usage.

There are few other ways to measure the results of approximation and thus measure an output error of a system. An Error Metric is used to measure the error of the output of the exact system and the approximate. It can provide a way for users to quantitatively compare the performance of competing systems or their representation. General-purpose error metrics are [16]:

- Arithmetic error metrics
  - Worst-case error, $e_{wst}(f, \hat{f}) = max_{\forall x \in B^n} |int(f(x)) - int(\hat{f}(x))|$
  - Mean absolute error $e_{mae} = \frac{1}{2^n} \sum_{\forall x \in B^n} |int(f(x)) - int(\hat{f}(x))|$
  - Relative worst-case error. $e_{rwe}(f, \hat{f}) = max_{\forall x \in B^n} \frac{|int(f(x)) - int(\hat{f}(x))|}{int(f(x))}$
- Generic error metrics
  - Error probability (Error rate) $e_{prob}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in B^n} [(f(x)) \neq (\hat{f}(x))]$ and
  - Average Hamming distance $e_{hd}(f, \hat{f}) = \frac{1}{2^n} \sum_{\forall x \in B^n} \sum_{i=0}^{m-1} f(x) \oplus \hat{f}(x)$.

where B is a set of available inputs, x is an input, $f, \hat{f}$ are functions that we want to compare, and $int()$ is a function that transforms the output to specific numeric representation.

Chapter 3

Evolutionary algorithms

From the history we know that it has always been big lust and challenge for computer engineers to develop something that is created automatically by computer but showing features of an innovation human design. As the discipline of AI evolved through years, many computer scientists realized that the best way to move forward, is to look to the basis of nature and learn from it. For example, neural networks are inspired by functioning of an actual human brain, Ant Colony Optimization (ACO) [5] algorithms are copying behaviour of an ant colony for solving hard optimization problems and evolutionary algorithms are inspired in biological principles.

3.1 Principle of evolutionary algorithms

If we look at what types of problems are solved in the field of artificial intelligence we could say that many of the tasks are solved by searching for a target solution in the space of possible solutions. This space can have many features where usually the size is the most important one. In computer science the crucial aspect is the search space size with respect to the computing resources. For example, if we have a small search space and relatively huge computing resources we can evaluate all possible solutions of the problem and decide what the global optimum is. On the other hand, complex and extensive problems are usually solved by heuristic and stochastic methods. One of these methods is now called the evolutionary algorithm. First search algorithms inspired by biological evolution were introduced in 1960's and 1970's under names evolutionary programming, evolutionary strategies and Genetic algorithms. From the early nineties they were unified and called evolutionary algorithm and used as a valuable tool for solving hard optimization problems and thus finding ideal parameter settings in optimized systems. Lately they have been used for automated design of various systems and the method is known as genetic programming.

3.1.1 Optimization in context of evolutionary algorithm

In mathematics we can look upon an optimization as a task of searching for a global maximum or minimum of a function.

$$f : A \longrightarrow \Re \tag{3.1}$$

Where the min/max is denoted as $x^* \in A \mid f(x^*) \leq f(x), \forall x \in A$, resp. $x^* \in A \mid f(x^*) \geq f(x), \forall x \in A$, and set A is the search space, x is a candidate solution and function f is the so-called fitness function in terms of evolutionary algorithms [5]. The set of

candidate solutions is called the population. It allows the EA to apply a parallel approach for searching. Every new population is created via biology-inspired operations such as cross-over and mutation. which are applied on candidate solutions.

Alg. in 1 shows pseudo-code of a typical EA. At the beginning an initial population is created. Pseudo random number generator or some heuristics can be used for this purpose. The initial population contains a fixed number of individuals. During the EA cycle, every population P in a certain time is called the generation, and is evaluated via the quality function. It is common that the fitness of an individual marks its biological fitness. Bigger the quality, bigger the value. In every generation some individuals are chosen and set as parents. This selection is based on individual's fitness value. Higher the score of an individual higher the chance to be selected as a parent. To create offspring, various operations are applied. Usually mutation and cross-over are most common. Final step is the selection of the next members of new population. They are selected from both sets of parents and offspring. There are many selection algorithms for example, roulette or tournament algorithm.

```
t = 0 ;
P(t) = createInitialPop ;
evaluate(P) ;
while ending condition is met do
    Q(t) = parentSelection(P(t));
    N(t) = createNewIndividual(Q(t));
    evaluate(N(t));
    P(t+1) = createNewPopulation(N(t),P(t));
    t++;
end
```

Algorithm 1: Pseudo code of evolutionary algorithm [a]

[a]Source: https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FEVO-IT%2Flectures%2F04-GenetickeAlg.pdf&cid=13232

3.2 Problem encoding

Problem encoding depends on a particular application. If the aim of EA is to optimize some parameters then a candidate solution is encoded as a sequence, for example, as floats, integers or binary numbers. If the aim is to evaluate programs such as in GP, then a candidate solution is represented as a syntax tree or graph. The encoded solution is called chromosome or genotype. While genetic operators work at the level of genotypes, the fitness function is applied on the so-called phenotypes which are created by means of a genotype-phenotype mapping from genotypes.

3.3 Mutation

Mutation is one of the most important operations in EA. It works with genes in the chromosome or a phenotype. It randomly selects a gene and changes its value. Naturally it depends on how the individuals are represented data-type wise. For example, in the binary representation, mutation inverts bits, in the integer operation it chooses a replacement value from a set of other integers etc. for example, like in the picture 3.3. We need to define

probability that determines how often this process is happening. In general, it is usually a small value, 0,1% for example. [1]

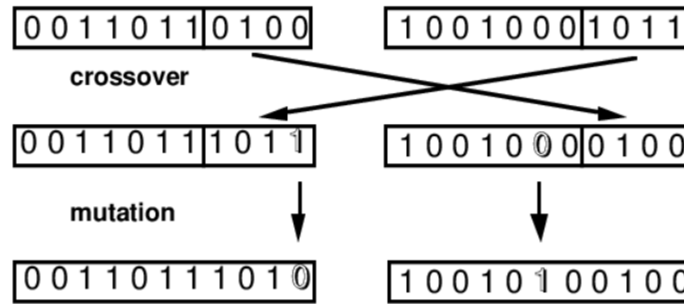


Figure 3.1: Example of binary mutation and crosover. [2]

3.4 Crossover

Crossover along with mutation is a very important operation as well. As shown in Fig. 3.2 It works with two chromosomes. Basic idea is in swapping interval of some genes between two chromosomes. There are several important types of crossover.

- single-point crossover,
- multi-point crossover,
- uniform crossover and,
- arithmetic crossover.

For example, we will now consider chromosomes in the binary interpretation and constant length. The Single-point crossover works this way: We choose one index in binary vectors and swap all bits after this index.

Figure 3.2: single point crossover

Multi-point crossover works in a similar way, but we define two or more crossover points and the bits between those points will be swapped. [3]

Figure 3.3: Multi-point crossover.

In Uniform crossover, for each position in the offspring is randomly decided if the gene is taken from the first or the second chromosome.

[1] Source: https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FEVO-IT%2Flectures%2F04-GenetickeAlg.pdf&cid=13232

[3] Source: https://wis.fit.vutbr.cz/FIT/st/cfs.php?file=%2Fcourse%2FEVO-IT%2Flectures%2F04-GenetickeAlg.pdf&cid=13232

3.5 Selection mechanisms

The selection mechanisms choose individuals that will serve as parents in the next generation. It has to find balance between preferring individuals with a high fitness value and securing the diversity of the population. Available approaches differ in the selection pressure which has a big impact on EA convergence. The more selection pressure the faster convergence. However, we don't necessarily need too fast convergence because it could lead to a local premature convergence to a local optimum. Major selection algorithms are as follows:

- roulette-wheel,
- stochastic universal sampling,
- elite algorithm and
- tournament algorithm.

In the roulette-wheel selection, the probability of selecting k-th individual is $\frac{f(k)}{\sum_{i=1}^{n} f(k_i)}$, where $f(k)$ is the fitness of k-th individual and n is the population size. this is usually implemented as the roulette-wheel selection algorithm. We spin the wheel as many times as many parents are needed to be selected.

In stochastic universal sampling We use the same approach as in the roulette-wheel but the difference is that we only spin the wheel once, because it uses multiple pointers determining new parents after each spin. These pointers are equally distributed around the wheel shown in fig 3.4.

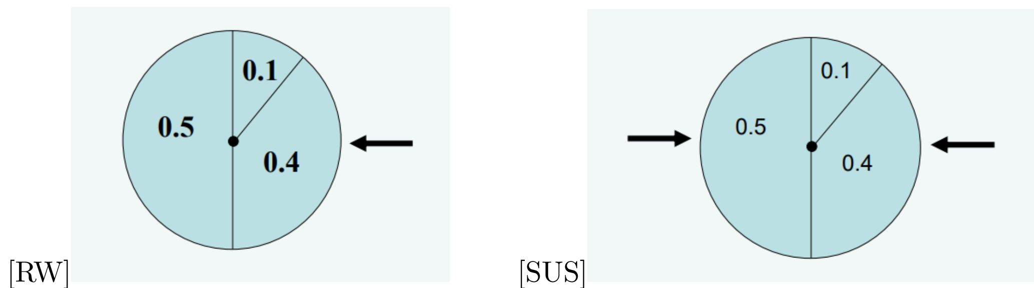


Figure 3.4: Roulete-Wheel and stochastic universal sampling algorithm.

The principle of the elite algorithm is to sort the individuals according to their fitness values. Only a certain number of best-scored individuals then is deterministically chosen.

Tournament relies on a certain (predetermined) number of individuals randomly selected from the population. The best rated individual in this group (the tournament winner) is selected as parent. This process is repeated until required/needed number of parents is selected.

3.6 EA in practice

When designing an EA application there are many aspects that have influence on it's behaviour and results of the evolution. They should be taken in consideration while developing EA strategy, and these aspects are:

- encoding,
- genetic operators and search strategy,
- fitness function,
- population size,
- initialization,
- number of generations or the termination condition and
- probabilities.

If we want to achieve good results we have to design these parameters very carefully. Other parameters are subject of experimentation.

Let:

- G be number of generations,
- P population size,
- T_a candidate solution evaluation time and
- T_p time to create a new population.

Then, the time of evolution is equal to $G * (P * T_a + T_p)$ and the number of fitness evaluations $G * P$ is usually considered.

Chapter 4

Genetic programming

Before we can describe what a Cartesian genetic programming (CGP) is , we will briefly introduce what the genetic programming (GP) is in general. Genetic programming is a machine learning technique which can be used as a tool for evolving some sophisticated system or developing better parameters for the system using evolutionary computation. GP was introduced in the 1980's and popularized by Jan Koza in the 1990's. Koza's GP uses tree-based representation where the nodes of a tree are functions and leaves can be substituted by constants and terminals. The root of a tree represents output of a program and the set of functions and terminals together is called a primitive set [14]. GP was not designed only for improving parameters of a programs but even for automatic generation and composing of programs. It is true that GP itself is inspired by EA a lot, but it differs in some main functionalities [14].

- Representation - in EA, individuals are usually represented as simple data structures, as array of integers or boolean values etc., but in GP individuals are represented as tree structures which replace dedicated programs.
- Operators - GP utilizes specialized genetic operators that operate directly on candidate programs.
- Fitness function - When calculating the fitness value for a candidate program is executed for defined set of inputs and then fitness score is calculated. results.

One of the main applications of GP is symbolic regression. It is a problem for finding a function that can approximate data with minimal error.

4.1 Cartesian genetic programming

Cartesian genetic programming (CGP) was firstly introduced in 1999 by Miller and Thomson paper [8]. In CGP the individual is not represented by a tree structure but as a directed acyclic graph [14].

4.1.1 Method of CGP

In this section we will demonstrate the usage of CGP in evolutionary design of combinational circuits. A candidate circuit is represented in a graph as an array of programmable nodes where there are n_c columns and n_r rows. At the beginning of an evolution the number of

inputs n_i and outputs n_o is determined. Each node performs one of the functions acquired from function set Γ. Every element can have up to n_n input arguments. Inputs of the individual elements which is in $i-th$ column can be connected into circuit's input or to the output of another element that is situated in one of $1, ..., L$ of previous columns. Parameter L is determining the scale of connectivity of the elements in the circuit. If its value is equal to 1, we can only connect layers that are neighbours and the connectivity is minimal. On the other hand if we choose this value equal to n_c maximum connectivity is enabled. The primary Inputs are not bound by this limitation and can be linked to elements in any column. The candidate solution, in other words, the chromosome contains A_{CGP} integer values [14]:

$$A_{CGP} = n_r n_c (n_n + 1) + n_0 \tag{4.1}$$

We encode a candidate solution in such a way that the inputs of program are given integer indexes in range $0, \ldots, n_i - 1$. The individual elements outputs are labeled in the same way but we begin with an element in the top left corner with value n_i then we continue incrementing this value for elements down in this column and in all next columns until we reach the output of the graph. The function representing an element will be labeled with integer as well. For every function in set Γ we will substitute its indexes for string value ($xor = 0, nor = 1, and = 2$). If an element has 2 inputs, for example, its encoding contains is 3 integers, as shown in Fig. 4.1. First element is representing function nor which is labeled with 1. Hence the third value of its chain will be $_, _, 1$. We can see that its inputs are 1 and 2 so its encoding is $(1, 2, 1)$. The Last two numbers of the chromosome define which nodes lead their outputs to graph output [14].

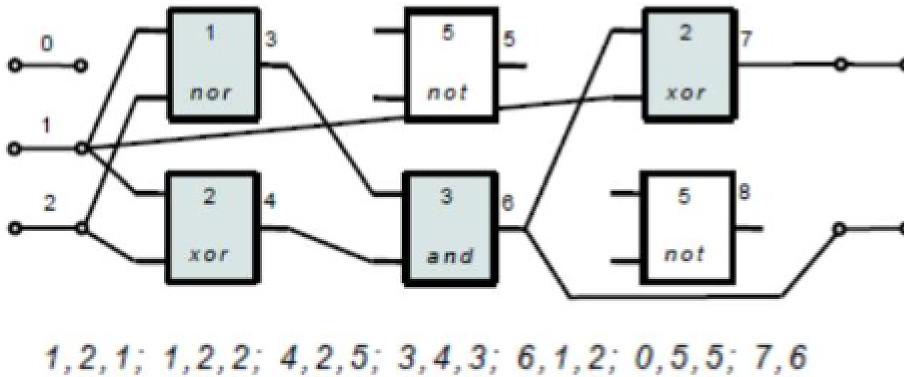


Figure 4.1: Example of a candidate circuit with its chromosome representation. [1]

The chromosome has some important properties.

- For every chromosome in every iteration its length is constant. On the other hand the size of the represented circuit (phenotype) can differ in each iteration.
- It is possible that some nodes do not have to be used in phenotype and some can be used multiple times.

- The same is valid for inputs of the elements.
- The Same is valid for the primary inputs.

While using a mutation in the chromosome it is possible to disconnect inputs of an element and we can also disconnect the entire function of an element. Of course this is happening randomly and we have to set the number of mutated genes. We have to ensure, while applying mutation, that the randomly created integers are valid [14].

4.1.2 Search method

The search algorithm is $ES(1 + \lambda)$. Population consists of $1 + \lambda$ offspring. We choose one individual with the best fitness and create its λ offspring. There is one important rule in CGP. If there are more individuals with best fitness we always choose the one which hasn't been served as parent in previous generation. With this approach we increase divergence in the population. Initial population is generated randomly or using some heuristics [14].

Result: p, fitness(p)
P = createTinitialRandomPop() ;
evaluate(P) ;
p = highest_scored_individual(P) ;
while *ending condition is met or out of loops* **do**
  α = highest-scored-individual(P) ;
  if $fintess(\alpha) \geq fitness(p)$ **then**
    p = α ;
  end
  P = create λ offsprings of p using mutation ;
  Evaluate(P) ;
end

Algorithm 2: CGP search method [a]

[a] Source: http://www.fit.vutbr.cz/~sekanina/publ/ahs18/ahs18_tut_approx_comp_1.pdf

4.2 Properties of CGP

When we discuss in this report that mutation is the only operation used in CGP it is clear that it has a significant influence on the whole search process. With mutation, active nodes can become inactive or active in the other state. Mutation can have zero impact on the fitness value of an individual in the following cases:

- An individual becomes another individual in the population with the same fitness because the mutation performed an operation on some active elements with no impact on the fitness value.
- Phenotype stays unchanged because the mutation was applied to new active genes.

Mutations are usually classified as adaptive, harmful or neutral. If we have a sequence of passive mutations followed by adaptive one, it may have a great positive influence on the individual fitness. We are mentioning this example because we can emphasize that mutation in this case substitutes another operation, crossover, which has the potential to significantly change the phenotype [14].

4.3 Optimization of CGP

It is crucial to optimize our implementation of CGP because within a range of experiments millions of evaluations are done when searching for a target solution. Naturally we want to make CGP evaluation as fast as possible, mainly the evaluation of candidate circuits. Let us suppose 3 inputs, 6 nodes and 2 outputs realizing a function as shown in 4.1. We want to find a correct circuit implementation with respect to a target truth table. Then we simply simulate them forward way. We have to run the simulation with 2^{n_i} test vectors to cover all possible combinations. It means the execution time will grow exponentially with the input vector size.

As the sequential circuit simulation is not scalable, we have to dive into parallel simulation. Parallel simulation uses binary operations existing in programming languages like in C. There exists instructions that can execute bit-wise operations in one processor clock cycle. And it is multiple times faster than the sequential approach.

Because today we usually work with 64 bit processors we can simulate in one clock cycle 64 input vectors. In general, we can accelerate circuit simulation $2^{n_t}/b$ times where b is number of bits in bit-wise logic instructions [14].

Chapter 5

Non-local means denoising filter

In image processing, we may obtain some side effects that are not wanted. These errors (noise) may be caused, for example, by random movement of the photon captured by sensors or amplified while digital processing of the image etc. Tools to handle these unwanted effects are called filters and they are used mainly for suppressing high or low frequencies, smoothing the image or detecting edges in the image [1].

5.1 Basic idea of non-local denoising

The basic idea of this type of filters is quite simple. Every pixel contains the rate of the three observed RGB colors values. Basically we take pixels in the defined neighborhood around this spot, calculate average values of these pixels and then replace the original value with the new one. A new approach is called non-local denoising filter. The idea is that if we can find elsewhere in the image pixels with a very similar neighbourhood we can denoise this value with an average value of these pixels. The problem is that these pixels don't have to be anywhere near to themselves.We can point out examples of this situation like when the picture has the same multiple objects in a row, periodic patterns or the elongated edges. Therefore, it's needed to scan a big percentage of the picture in order to find these pixels [1]. Basic idea of a non local denoising is shown in Fig. 5.1, and the results of this filter can be seen in Fig. 5.2. [1]

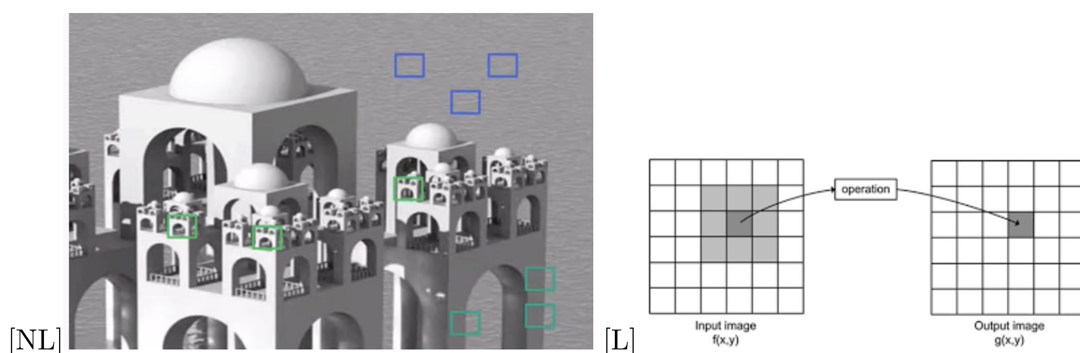


Figure 5.1: Difference between Non-local filter (left image) and local filter (right image).

[1]source: http://what-when-how.com/introduction-to-video-and-image-processing/neighborhood-processing-introduction-to-video-and-image-processing-part-1/

Figure 5.2: Left image: original picture. Middle image: noised picture. Right image: denoised picture [1].

5.2 Non-local means filter

The objective is to find pixels with similar neighborhood in a window with predefined size. In other words it says we are looking for similarities in predefined space around a point and not in the whole image. This approach can be formulated as:

$$NLu(p) = \frac{1}{C(p)} \int f(d(B(p), B(q)))u(q)dq, \tag{5.1}$$

where $d(B(p), B(q))$ is a measurement unit describing Euclidean distance between two badges represented by B centered in pixels p, q. f is a function that has decreasing behaviour. C(p) is the normalizing factor and the $u(q)$ is pixel color value. In this example, it is shown that the filter works as non-weighted one. Which means that pixels are treated with the same importance and according to that we can calculate denoiseing values for all pixels in $B(q)$.However we use an exact NL-filter in this report the employs weighted average for denoising purposes as shown in Eq. 5.3 [1].

5.3 Pixelwise implementation

Consider that we have a color image $i = (i_r, i_g, i_b)$, where $i_k | k \in R, G, B$ are RGB layers, and pixels p, q in this picture. Then we are trying to calculate new pixel value $\hat{u}_i(p)$

$$\hat{u}_i(p) = \frac{1}{C(p)} \sum_{q \in B(p,r)} u_i(p)w(p,q) \tag{5.2}$$

$$\frac{1}{C(p)} = \sum_{q \in B(p,r)} w(p,q) \tag{5.3}$$

where $B(p, r)$ is a metric that indicates a limited window centered at pixel p with size $(2r + 1) \times (2r + 1)$. The neighborhood is limited to size 21×21 or 35×35 depending on value σ which represents standard deviation and it is chosen as a parameter before filtering. This search window is used for finding more pixels with similar neighborhood with size 3×3 respectively 5×5, see table 5.2. The filter uses weighted averaging for denosing purposes. The calculation of weights is shown in Eq. 5.5 and it uses squared Euclidean distance of

the $(2f+1)(2f+1)$ color patches centered respectively at p and q which is calculated as: [1]

$$d^2 = d^2(B(p,f), B(q,f)) = \frac{1}{3(2f+1)^2} \sum_{i=1}^{3} \sum_{j \in B(0,f)} (u_i(p+j) - u_i(q+j))^2 \tag{5.4}$$

For calculating the weights, the filter uses exponential function that is captured in Eq. 5.5, where σ is standard deviation value of a noise, h is a parameter with value dependable on a deviation.

$$w(p,q) = e^{-\frac{max(d^2-\sigma^2,0.0)}{h^2}} \tag{5.5}$$

The weight function is set in order to average similar patches up to noise. That is, patches with square distances smaller than 2^2 are set to 1, while larger distances decrease rapidly accordingly to the exponential function.

5.4 Patchwise Implementation

In this section we are going to discuss how a patch is denoised. Image stays defined identically as in section 5.2. Denoising is performed according to Eq. 5.6 [10].

$$\hat{B}_i = \frac{1}{C} \sum_{Q=Q(q,f) \in B(p,r)} u_I(Q) w(B,Q), C = \sum_{Q=Q(q,f) \in B(p,r)} w(B,Q) \tag{5.6}$$

Where $i, B(p,r), w(B(p,f), B(q,f))$ and size $(2r+1) \times (2r+1)$ represents the same variables as in Eq. 5.2. If we apply this on all patches we will get possible outcomes for all pixels and these estimates will lead us to final computation of denoising a picture with equation 5.7 where N denotes similar meaning as C in Eq. 5.2 [1]

$$\hat{u}_i(p) = \frac{1}{N^2} \sum_{Q=Q(q,f) | q \in B(q,f)} \hat{Q}_i(p). \tag{5.7}$$

We can compare patchwise and pixelwise implementation. The patchwise implementation seems to have better gain in terms of PSNR (peak signal-to-noise ratio) between original image and denoised image. However in terms of detail preservation the pixelwise implementation seems to be a better approach [1].

5.5 Software implementation

The method described in [1] is accompanied by software implementation [2]. Program is written in C and takes four arguments.

- σ - standard deviation value for noising an input picture and setting up window sizes by Tab. 5.1.
- path to an input image.
- path to a file where noised image will be stored.

[2]https://www.ipol.im/pub/art/2011/bcm$_n lm$/

- path to a file where denoised image will be stored.

AS it was described in section 5.1, the NL-means searches in a defined area for similar patterns. The size of this area as well as the size of a neighbourhood (patch) of a pixel depends on a σ value. If the standard deviation value is high, it is needed to make a bigger patch because we need to make the patch comparison robust enough. With that we need to increase a search space as well to increase the noise removal capacity of the algorithm by finding more similar pixels. The filtering parameter used in Eq. 5.5 is calculated as $h = k\sigma$. Also the size of patch is increasing, value of k is decreasing. Typical Values of parameters and its combinations are shown in table 5.1 and 5.2

σ	patch	pixel neighborhood	h
$0 < \sigma \leq 15$	3×3	21×21	0.40σ
$15 < \sigma \leq 30$	5×5	21×21	0.40σ
$30 < \sigma \leq 45$	7×7	35×35	0.35σ
$45 < \sigma \leq 75$	9×9	35×35	0.35σ
$75 < \sigma \leq 100$	11×11	35×35	0.30σ

Table 5.1: Parameters for patchwise implementation processing grey image [1].

σ	patch	pixel neighborhood	h
$0 < \sigma \leq 25$	3×3	21×21	0.55σ
$25 < \sigma \leq 55$	5×5	35×35	0.40σ
$55 < \sigma \leq 100$	7×7	35×35	0.35σ

Table 5.2: Parameters for patchwise implementation processing colored image [1].

5.6 Approximation computing in image processing

In paper [3] it was shown that 83% of computation kernels are suitable for approximation. For example, one of the most contested parts were a dot product computations and distance computations. In many papers about image processing is notable that elementary arithmetic circuits are a great target for approximation. There are two important types of approximations presented in literature. Let us consider circuit-level approximations. These approximations can be conducted as:

- Aproximation of smaller units in circuits like adders multipliers or comparators.
- Higher level of a component such as filters, DCT and FFT created using these component.

It seems that simple filters such as median filter can be approximated as whole system. It can be decomposed into scheme only containing comparators and registers. In NL-means filter it is rather more difficult problem [13].

5.7 Approximation approaches

In this section we describe two approaches for approximation of an image filter circuit. First approach is built on a principle of approximating implemented circuit, in which we can remove some components and reconnect others in a different way. But we have to take in consideration that approximate implementation should not affect the error significantly.

The second approach does not uses any existing implementation and tries to build a circuit from scratch. For this purpose we have a set of suitable components (Comparators, adders, etc.) and rules, how components can be connected in the target circuit.

Then we will calculate an error using some Error metrics, for example, mean absolute error (MAE) between outputs of candidate filter f and golden filter g.

$$MAE = \frac{1}{K}\sum_{i=1}^{K}|O_f(i) - O_g(i)| \tag{5.8}$$

where K is the number of pixels [13], O_f, resp. O_g are output values of filters f, g.

5.8 Approximation of a NL-means filter

NL-means filter is impossible to approximate as whole unit.Hence our objective is to find a component or operation in the NL-means filter that could be suitable for approximation. We decided to approximate the multiplication operation as it is used very often in this filter. As NL-means filter could be applied in embedded systems, we will perform approximation at the circuit level. Table 5.3 shows percentage representation of numeric operations performed in the NL-means filter filtering a picture with resolution 930 $\times$ 632 pixels and noising parameter $\sigma = 50$. These numbers were obtained by using a filter implementation from [1]. The multiplication operation represents nearly half of all computations. This is the reason why the focus of the thesis will be approximate implementation.

Arguably of course for every image and sigma value the percentage is different but experiments shown that only very small images with low sigma settings achieve percentage of multiplication 33%. Which is still big amount.

type	percentige	count
multiplications	48.84%	207262430568
additions	25.59%	108623850310
subtractions	25.22%	107047289676
division	0.35%	1487417008

Table 5.3: Percentage share of numeric operations when filtering a 930 $\times$ 632 pixels picture and $\sigma = 50$.

The next important question which needs to be answered is which multiplications of the original source code [1] will be approximated. The answer is that we will approximate all the multiplication necessary to meet the filter operation. In the implementation of the filter [1] there are lot's of auxiliary multiplications which do not directly contribute to the filter calculations. They include, for example, i/o png images handling, settings of some constants, prepossessing of a picture. The following list summarizes the places where the approximate multiplication will be introduced. Please note that each RGB channels is handled separately.

- From Eq. 5.6 Patchwise implementation $u_I(Q) \cdot w(B, Q)$.
- From Eq. 5.5 d^2, not including the σ^2 and h^2 because they are a constant value.
- From Eq. 5.4 $(u_i(p + j) - u_i(q + j))^2$, not the $3(2f + 1)^2$ because it is an integer multiplication.
- From Eq. 5.2 Pixelwise implementation $u_i(p) \cdot w(p, q)$.

So basically approximate multiplication is implemented at all places in all math formulas in which a floating point multiplication is used, including calculations on the power of two. Keep in mind that table 5.3 was created in such way that we considered only the multiplications conducted at the preselected places of the source code.

Of course the software implementation of this filter is not the only thing of interest. Also a possible hardware implementation and its approximation is the goal of this thesis.

Chapter 6

Floating point to fix point arithmetic

The image filter described in section 5 uses IEEE 754 representation of floating point data type for its computations. In the standard, floating point data type is represented in 32 bits and it is composed of three separate components. If we look at this representation in a specific way, floating point number can be viewed as a $mantissa \times base^{exponent}$. This structure can be more described in bot-wise preview where:

- The sign bit determines if the Mantissa is positive or negative number (1 bit width).
- The bias is basically a number, that is added to the exponent. The reason why we use bias is that we need to represent both positive and negative numbers (8 bits wide).
- The mantissa is always a number $1..x..x$ in normalized form, i.e. therefore there is no need to represent the leading 1. (23 bits).

The fixed point number representation is an attractive alternative to floating point emulations. According to [15] an 8-bit fixed-point integer multiply consumes 15.5× less energy (12.4× less area) than 32-bit fixed point multiply, and 18.5× less energy (27.5× less area) than a 32-bit floating point multiply. C language does not support arbitrary fixed point data types. Therefore, it is a programmer task to effectively represent non-standard integer data types and convert them to a float and vice versa. Even a bigger problem is to apply some arithmetic operations like multiplication or division while using this type of representation. Specific approach for the conversion is to use bit-wise operations and store fixed point representation of a floating point number into integer variable. The list of problems in general with this approach is following [4]:

- When multiplying, it is highly likely that the result has to be stored in more bits than the two operands are.
- The precision can be lost, the proportion of the loss depends on circumstances.
- It is unclear to determine optional properties for shifting operations due to compromise trade off between quality and functionality with respect to an application.

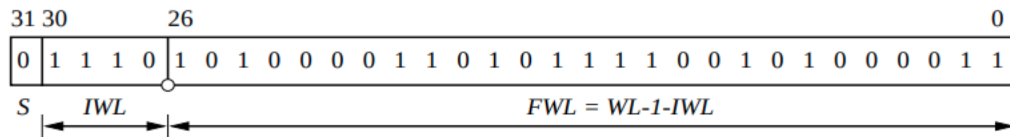


Figure 6.1: Example of a signed fixed point number bit-wise representation. WL=32, IWL=4, FWL=27.

6.1 Fixed point representation

If we take an integer value and multiply it with power of two and on exponent, we can look at it as fixed point representation of floating point number. In the simplest way, we can say that we pick an index position of a bit in this bit sequence and put a decimal point right after the index value. The right part represents fractional bits and the left part represents whole (integer) part of a value. Of course we have to reserve the Most significant bit (MSB) to store the information about the number sign. In general, a number represented in the fixed point arithmetic has these attributes (layout) [4]:

- Word length (WL) represents bit length of a whole integer.
- Integer word length (IWL) stands for a bit length of an integer part.
- Fractional word length (FWL) is length of fractional part.

The layout has a big impact on ranges in which we can store floating point values. For example, if we look at Fig. 6.1 we see that the integer part has only 4 bits, meanwhile fractional part has 27 bits. This means that the number range we can store in the integer part is $< -15, 15 >$. But this applies only on integer part. In fractional part we have much bigger range that we can store. Number that is visualized in fig 6.1 is 14.631578947368421 . The accuracy of fractional part equal to 13 decimal digits.

It is vital to realize that FWL can be computed as follows $FWL = WL - IWL - 1$ [4].

6.2 Fixed point arithmetic operations

In this section we are focusing on how fixed point arithmetic operations can be performed. What we need to take into consideration is mainly size of both operands with the respect to the format of the result.

Addition and comparison are easy to implement and nothing is really changed apart from classic floating point arithmetic. The only difference is result size, which is determined this way [4]:

$$IWL_r = \max\{IWL_a, IWL_b\} \tag{6.1}$$

Where a, b are two integer numbers and IWL_a, resp. IWL_B is bit-length of number a resp. b. Note that equations in Eq. 6.1, 6.2 and 6.3 shows only result number bit-width not the actual arithmetic results. For example, if we had number a, with bit-width equal to 10 and number b, with bit-width equal to 5 and used equation 6.1, then the result would have bit-width equal to 10. Division is little bit trickier because we can still use a common integer divider, but the size of the result is

$$IWL_r = \text{ WL } - 1 + IWL_a - IWL_b \tag{6.2}$$

Multiplication is quite difficult as well as division to perform and there are few approaches that will be mentioned. The problem is that the size of the result has to be stored mostly in $2 \cdot WL$ bits and thus: [4]

$$IWL_r = IWL_a + IWL_b + 1 \tag{6.3}$$

So it follows that in the most processors or multipliers, The product is stored into WL bits and upper bits of a result are cut off (an overflow happens). This is happening because in most high-level languages the compiler maps a result of two source operands into the same data type. Of course this causes poorly calculated results.

In this chapter we address three solutions to how to deal with this problem. These approaches are considered well functioning, but they differ in advantages and disadvantages.

1. Make the upper word of the result accessible, meaning casting operands to $2WL$ bit width.
2. Using shifting operations to scale operands to the power of 2. The result will fit in WL bit width.
3. Use the macro for obtaining on the upper part as multiplication implementation.

As we can see, the first option's advantage is that the result will have great accuracy and it is probably the most precise option among these three. But the problem is that embedded systems are, in most cases not built this way. Imagine a 32 bit architecture. Casting 32 bit integers onto 64 bit integers is in many cases forbidden or impossible because 32 bit systems are not build to deal with 64 bits integers. Even if it is allowed, it is much slower. Imagine making multiplication on 64 bits and afterwards regulating the result with 64-bit shifting operation, which you have to do, as well [4].

The second option provides us option trading off speed for accuracy. This method uses shift operations. Before multiplying we shift both operands to the right. Then we multiply and the result will be again shifted to the right. It has a condition. The number of shifts has to be equal to $\frac{WL}{2}$. But changing the size of the shift operation is giving us a chance to trade off range for precision. For example, imagine 32 bit fixed point numbers with $IWL = FWL = 16$ and two numbers $7.287, 3.0$. Next we apply two different shift operations, compare the result of these two multiplications of numbers x and y. [1]

$$(float)7.287 * (float)3.0 = (float)21,861 \tag{6.4}$$

$$(((x >> 8) * (y >> 8)) >> 0) = 21,83203125 \tag{6.5}$$

$$(((x >> 4) * (y >> 4)) >> 8) = 21.83349609 \tag{6.6}$$

Why are these results different? If we shift operands by 4 bits like in equation 6.6, we will lose 2 bits in fractional part in comparison with the example in Eq. 6.5. On the other hand, if we use Eq. 6.5 we will not gain two more bits for the integer part, as it is in Eq. 6.6. That is why Eq. 6.5 is little bit more precise.

[1] Example and principles take over from video: https://www.youtube.com/watch?v=S12qx1DwjVk&list=LL&index=17&t=1129s

But if we change numbers to let us say 128.0 and 3.0, a problem occurs. If we use Eq. 6.5 with numbers 128 and 3 the result will be -128 which is clearly not correct. It is because we don't have the range in the integer part. But if we use Eq.6.4, the result will be 384, which is correct, and then the range is clearly sufficient enough.

The problem with this approach is that it is highly not accurate. And it is working only on the limited range as well.

6.3 Experiments with bit-width and fixed point representation

The goal of following experiments will be to investigate to what extend the precision of arithmetic operations influences the quality of the NL-means filter. In embedded systems, the use of the fixed point arithmetic instead of the floating point arithmetic one is often requested to reduce resources. The main focus will be on reducing bit width for multiplication.

Figure 6.2: Original.

Figure 6.3: Noised with $\sigma = 50$.

Figure 6.4: Denoised picture by original implementation of the filter from chapter 5.

In figure 6.2 we can see a reference image for our experiments [2]. Fig. 6.2 shows a corrupted image with σ noise parameter equal to 50 and in Fig 6.2 is the result of denoising using NL-means filter. First matter before we try to manipulate with implementation of multiplication operation, is that we have to find the range of numbers that are entering a multiplier. If we can determine values that are statistically important for calculations in a filter, we would be able to aim for much better solution of approximation than trying some random solutions.

Fig. 6.6 shows histogram of operands entering multiplication operations in the NL-means filter which was applied on the image shown in Fig. 6.2. Note that the filter works with floating point representation of numbers. The reason why the histogram shows only integer number representation, is that in this application there are too many multiplication operations to perform, thus it wouldn't be possible to visualize every floating point number in a single graph. Hence the solution is to normalize the histogram into integer values. This means that a specific integer value represents all floating point numbers with the same integer part and independent of its fractional part.

Furthermore it shows us the the majority of operands entering multipliers is within the range $(-200, 300)$. Most of them are more or less between $(-1, 1)$. But of course this is not everything. The entire range of values was $< -540, 560 >$. The question is which numbers are statistically important for calculations. This question will be answered further in this section.

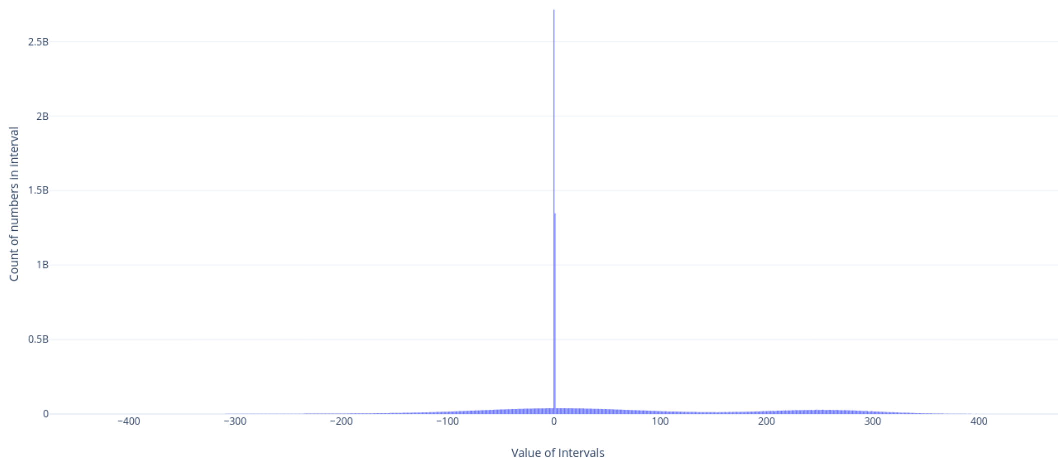


Figure 6.5: Histogram of operands entering multiplication operations in the NL-means filter which was applied on the image shown in Fig. 6.2. For simplicity float values are rounded to integer values so one integer value is representing count of all floating numbers in its range (in billions).

[2]Source of the image https://fangohr.github.io/computing/imagemagick.html

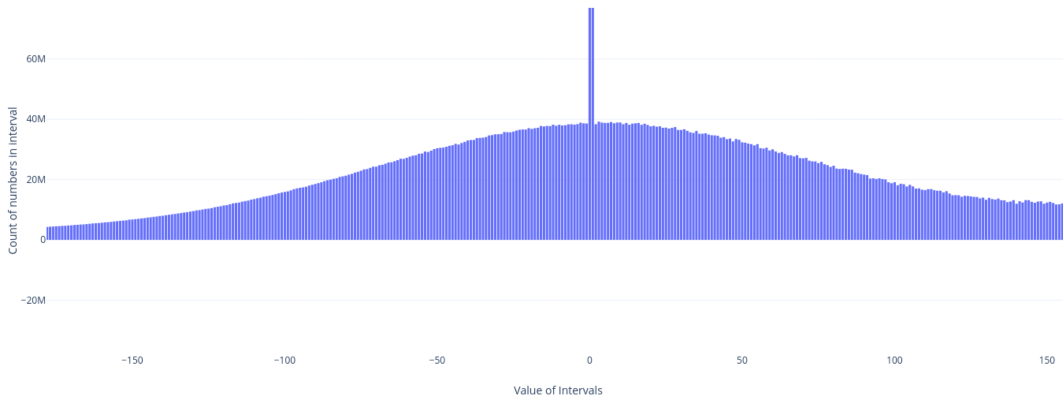


Figure 6.6: Zoom of the histogram from 6.6 to provide detailed view.

6.4 Casting operands to double bit width

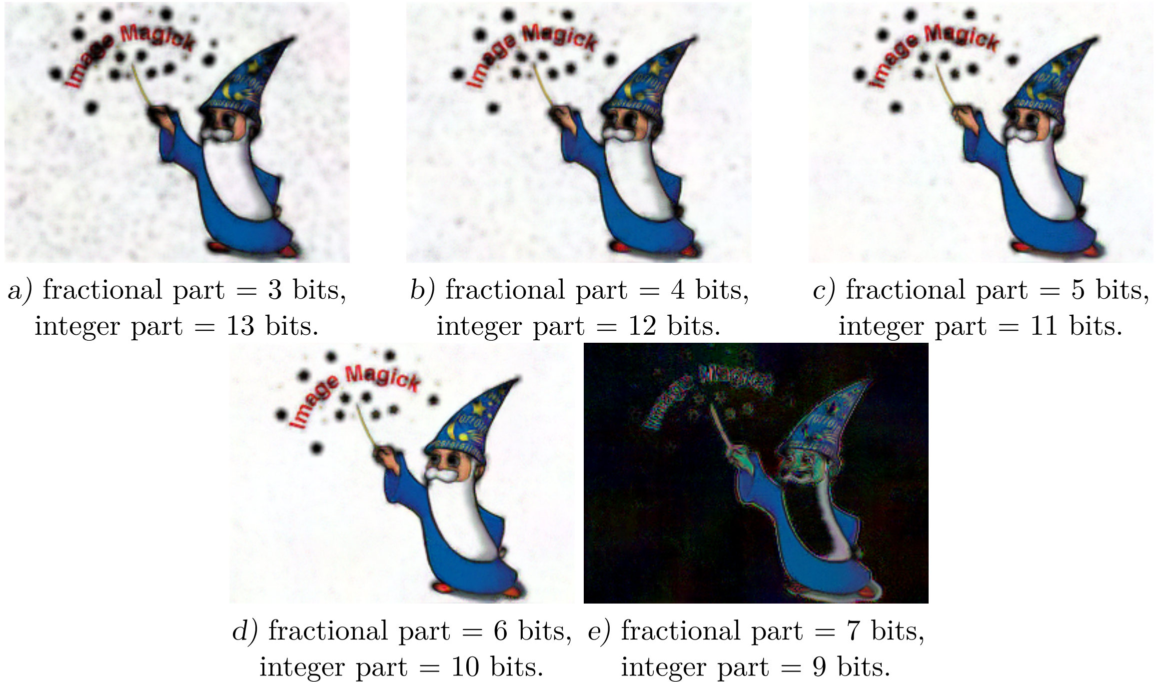
Next set of pictures in figs 6.7, 6.7,6.9 and tables in 6.1, 6.2 and 6.3 will show the results of the experiments. We will be looking at visual differences of the filtered images and the corresponding quality metrics. First we will be comparing the fixed point conversion technique with casting operands to $2WL$ size. Of course two options (16 and 32 bits) are considered. The question is why not use less bits and the answer is that the experiments shown that 10 bits are the minimum we have to use for integral part. If the range is lower, then the quality of a picture is not acceptable. See the resulting images filtered on 32 bit architecture in figure 6.7 and 6.7. This is why we cannot, for example, use the 8 bit arithmetic.

Figure 6.7: bit-length = 32, integer part = 10 bits, fractional part = 22 bits.

Figure 6.8: bit-length = 32, integer part = 9 bits, fractional part = 23 bits.

When we have a 16 bit architecture for fixed point representation another question arises. What is the ideal distribution of bit width ranges between IWL and FWL. For example, if we use 10 bits for the integer part, only 5 bits remains for its fractional part, which could be insufficient. Of course the more bits we use for fractional part the more precise the result will be. Pictures in figures 6.9 show denoised results with a different

number of fractional bits. In variant *a*) only three fractional bits are used. Anyhow it can bee seen that there is too much noise still present and the picture is not really anymore close to the original image. On the other hand, when 6 fractional bits are used shown in *d*), the result is much better but there is still a visible difference in some parts of the image. Finally, in the picture *e*), we can see the same result as with 32 architecture which is that alreadz mentioned minimum of 10 bits is required int the usage in integer part. This leaves us in the conclusion that in 16 bit architecture the best choice is to use $IWL = 10, FWL = 6$.



a) fractional part = 3 bits, integer part = 13 bits.
b) fractional part = 4 bits, integer part = 12 bits.
c) fractional part = 5 bits, integer part = 11 bits.
d) fractional part = 6 bits, integer part = 10 bits.
e) fractional part = 7 bits, integer part = 9 bits.

Figure 6.9: Results of using 16-bit arithmetic with different FWL and IWL settings.

Table 6.1 shows the results of quality metrics calculated from pictures in Fig. 6.9. Pictures from fig 6.9 were compared with original reference image 6.2. For the quality evaluation the Peak signal to noise ratio (PSNR) and Root-mean-square deviation (RMSE) metrics were chosen. Both these metrics use mean square error calculations. PSNR works in a logarithmic scale because many images(signals) have a wide dynamic spectrum, but not in this particular example. Higher the PSNR number is, the better quality. On the other hand RMSE shows us the square root of the differences between original values and denoised values or, in other words, the quadratic mean of these differences. the Lower the value is, the better the result.

FWL	3	4	5	6	7
PSNR	16.08	17.95	18.87	20.57	1.05
RMSE	40.05	32.30	29.05	23.87	226.04

Table 6.1: Selected metrics used for comparison between pictures in fig 6.9 and the original denoised image WL = 16.

	Original denoised
PSNR	27.06
RMSE	11.32

Table 6.2: Results of comparison between template image and original denoised image using floating point representation.

Of course these all calculations were performed on a half precision than in the original application. For illustration how the fixed point implementation is worse or better than the standard floating point arithmetic, we will show results performed on 32 bit architecture. The reason is that it is, important to check how big difference in quality is between these approaches, so in the end we can evaluate every aspect of the method.

a) fractional part = 4 bits, integer part = 28 bits. *b) fractional part = 16 bits, integer part = 16 bits.* *c) fractional part = 22 bits, integer part = 10 bits.*

Figure 6.10: Results of using 32-bit arithmetic with different FWL and IWL settings.

FWL	4	16	22
PSNR	17.98	26.65	27.22
RMSE	32.16	11.86	11.11

Table 6.3: Selected metrics used for comparison between pictures in fig 6.9 and original denoised image.

In Fig. 6.10 *b* and Fig. 6.9 *a* WL = 16 and FW value is set to 4. Visually, if we compare them, they look very similar and even if we look at the tables 6.1, 6.3 the numbers are almost identical. This is a clear evidence that it doesn't matter if we set the IWL value to 10 or 28. It points out that the only logical outcome is to only use values 10 or maybe 11 bits for the integer. The histogram in figure 6.6 only supports this claim.

Now if we look at figure 6.10 image *b)* and *c)* and corresponding values in table 6.3, the same occurence can be seen. These images look really alike, even if the FWL values are widely different. From these findings we could determine or predict what is the optimal FWL value. In graph 6.11, PSNR and RMSE values are ploted with respect to FWL. From this graph we can understand the impact of choosing any FWL value on the cost of cost/precision trade off. Specifically it shows that values 13 and above provide only minimal additional quality.

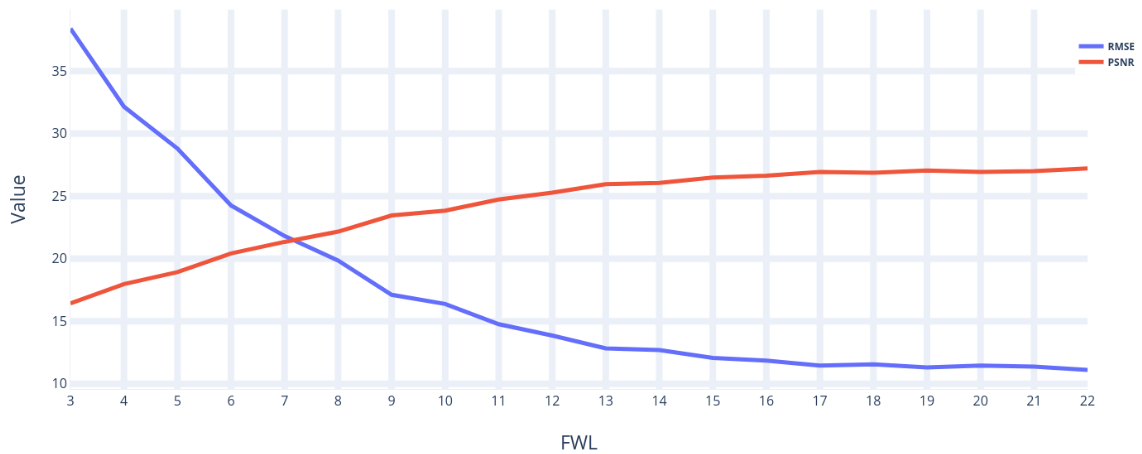


Figure 6.11: Results of using 32-bit arithmetic with different FWL and IWL settings.

6.5 Shifting operations in fixed-point arithmetic

In the previous section we described one of the safer, precise but costly way to perform conversion from floating point to fixed point. Now we will compare it with a less precise but less costly approach. As described in section 6.2 shifting operands operation does not need more bits to perform the conversion. It works with the original bit width of the operands. Of course this comes with the cost of less precision and directing the ratio between shifting result/operands. Because of this lost precision, we will show some result not only for 16 and 32 bit width but even for 64bit architecture as for better visual demonstration.

a) fractional part = 8 bits, integral part = 24 bits, shifting operands by 2 bits and result by 4 bits.

b) fractional part = 8 bits, integral part = 24 bits, shifting operands by 4 bits and result by 0 bits.

b) fractional part = 8 bits, integral part = 24 bits, with casting operands.

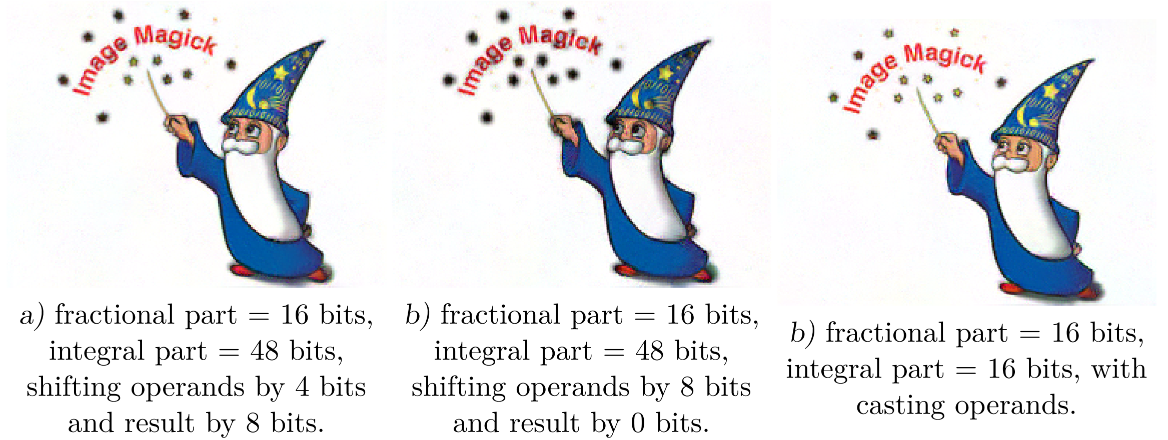
Figure 6.12: Results of using 32-bit arithmetic with different shifting settings.

As we can see in images 6.12 and in table 6.4 the best result is obtained by using the casting operands approach. As expected, both attempts to use shifting operations reports significantly higher error, and low image quality. If we compare different shifting strategies, shifting a result rather than not shifting the result seems like a better option. It is because if more we shift the bit representation of a value to the right, the more information we will loose. This is why image in Fig. 6.12 *a)* is slightly better than 6.12 *b)*. Of course changing

the operand shift value, has an effect on how the result shifting value will look like. It works both ways proportionally.

Conversion type	shift (op by 4, res by 0)	shift (op by 2, res by 4)	Casting operands
PSNR	17.94	20.22	22.12
RMSE	32.32	24.87	19.98

Table 6.4: The image quality obtained after applying different conversion approaches 'op' = (operand), 'res' = (result). WL = 32.



a) fractional part = 16 bits, integral part = 48 bits, shifting operands by 4 bits and result by 8 bits.

b) fractional part = 16 bits, integral part = 48 bits, shifting operands by 8 bits and result by 0 bits.

b) fractional part = 16 bits, integral part = 16 bits, with casting operands.

Figure 6.13: Results of using 64-bit arithmetic with different shifting settings.

Using even 32bit architecture with this approach showed that the results are quite inadequate. We could look upon a higher bit width and see if it makes a viable difference. The results for 64 bit architecture are shown in Fig. 6.13. As with 32 bit width, the visual and metric results in table 6.5 are not significantly better with respect to using 64 bits. But we need to take into consideration, that we just doubled bit width with premise to get better results and still, it is worse than using the casting operands technique with only 32bits

Conversion type	shift (op by 8, res by 0)	shift (op by 4, res by 8)	Casting operands
PSNR	22.46	25.10	26.47
RMSE	19.21	14.18	12.11

Table 6.5: The image quality obtained after applying different conversion approaches 'op' = (operand), 'res' = (result). WL = 64.

Finally Fig. 6.14 show results when using only 16 bit architecture. But this approach does not clearly lead to satisfactory image quality.

a) fractional part = 4 bits,
integral part = 12 bits,
shifting operands by 2 bits
and result by 0 bits.

b) fractional part = 4 bits,
integral part = 12 bits,
shifting operands by 1 bit
and result by 2 bits.

Figure 6.14: Results of using 16-bit arithmetic with different shifting settings.

Furthermore we can conclude that the shifting approach in this particular use case is not preferable. All fixed point conversions , due to its better parameters, in this image filtering task will be performed by the casting operands method.

Chapter 7

EvoApproxLib

In section 6.2 we described some basic bit width reduction techniques and how approximation of a circuit might be done. Of course these approaches are not particularly effective nor enhanced. But it showed us a way which we should look for. In this chapter we will be focusing on EvoApproxLib. This library contains many types of approximated adders and multipliers. As our task is to simplify the multiplication operations in the NL-means filter, the use of approximate multipliers from EvoApproxLib is a straightforward approach.

7.1 What is EvoApproxLib

The EvoApproxLib itself contains about 471 non-dominated 8-bit multipliers created from 6 convectional one. Also it has variations for signed and unsigned multipliers in $9 \times 9, 11 \times 11, 12 \times 12, 16 \times 16$ input bits. All these approximate circuit are available in C code, verilog, python and matlab. All multipliers are benchmarked and evaluated according to several error metrics. Also every circuit has information about its area and power consumption.

Because we will design approximate multipliers in chapter 8, it is important to mention that approximate multipliers from EvoApproxLib were created by a multi-objective CGP method. For representation a directed acyclic graphs were used. Via this approach it was possible to aim for minimization of a delay, power consumption and its error. Principles of CGP are described in chapter 4.1. For approximation of the multipliers they used this CGP parameters [11].

- $populationsize = 500, gen = 100000$.
- $n_c = 1000, n_r = 1$.
- mutation rate 5%.

Moreover EvoApproxLib contains also 28 exact multipliers. All error metrics that are subjected to the circuits are MSE (Mean square error), EP (Error probability), WCE (worst case error), MAE (Meean absolute error), MRE (Mean relative error).

7.2 Experiments with 16 bit approximate multiplier

The main reason why we discussed the fixed point arithmetic in section 6.3 is that EvoApproxLib circuits accept integer numbers as input. Hence the implementation of these approximated circuits works at bit level (using logical and shifting operations). in order to

safely simulate the behaviour of approximate multiplication in the NL-means filter, the approach is to convert floating point to fixed point arithmetic, feed approximated multiplier with this input and convert the fixed point result back into floating point data type which is used in the remaining part of the source code.

It was also matter of discussion what is the most efficient configuration (WL, IWL and FWL) for this particular problem. It was proposed that the best approach is setting IWL to 10 and FWL to 6 in 16 bit architecture. All calculations further in this section will be performed with this settings.

Circuit name	***MAE***	***WCE***	***MRE***	***EP***	**power**	**area**
mul16s_HEB	0.00 %	0.00 %	0.00 %	0.00 %	2.400	2614.0
mul16s_HDG	0.00032 %	0.0015 %	0.034 %	75.00 %	2.130	2576.5
mul16s_HFZ	0.002 %	0.011 %	0.22 %	98.43 %	1.483	1935.9
mul16s_G7F	0.013 %	0.0053 %	0.12 %7	87.50 %	1.961	2495.7
mul16s_GAT	0.012 %	0.048 %	1.06 %	98.44 %	1.396	1932.6

Table 7.1: Error metrics results for different types of approximated multipliers [11].

Table 7.1 shows the parameters of 4 different approximated multipliers with respect to error metrics, area and power consumption.

Circuit name	***PSNR***	***RMSE***
mul16s_HEB	20.12	25.14
mul16s_HDG	19.03	28.51
mul16s_HFZ	16.44	38.41
mul16s_G7F	16.54	37.96
mul16s_GAT	9.48	85.62

Table 7.2: Table showing PSNR and RMSE values for signed 16 bit approximated multipliers [11].

Fig. 7.1 provides us with the pictures to compare. They give us best information about qualities of the filtering. Visually, results can be sorted into two classes because clearly images in pictures *c-e* have a lot of visible noise. Applying multiplier *mul16s_GAT* devalued the image a lot. On the other hand pictures *a-b* are much better in this regard. There is a minimal difference between these two images produced by approximated multipliers and non-approximated ones. These visual outcomes are supported by quality measures in table 7.2, where we can see relatively large differences between PSNR and RMSE.

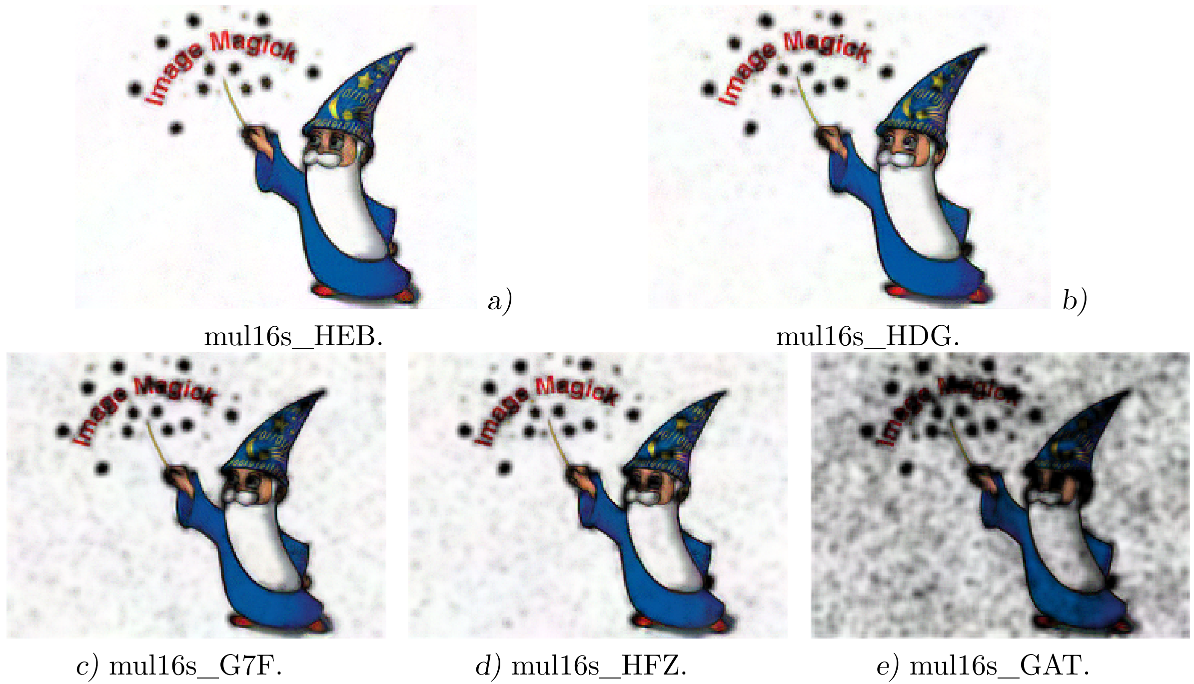


Figure 7.1: Results of applying different types of approximated 16-bit multipliers to NL-means filter implementation.

The question is what would be the best choice for using each type of approximated multiplier in this specific application. Of course the goal is to balance quality of filtering with the implementation cost on a chip.

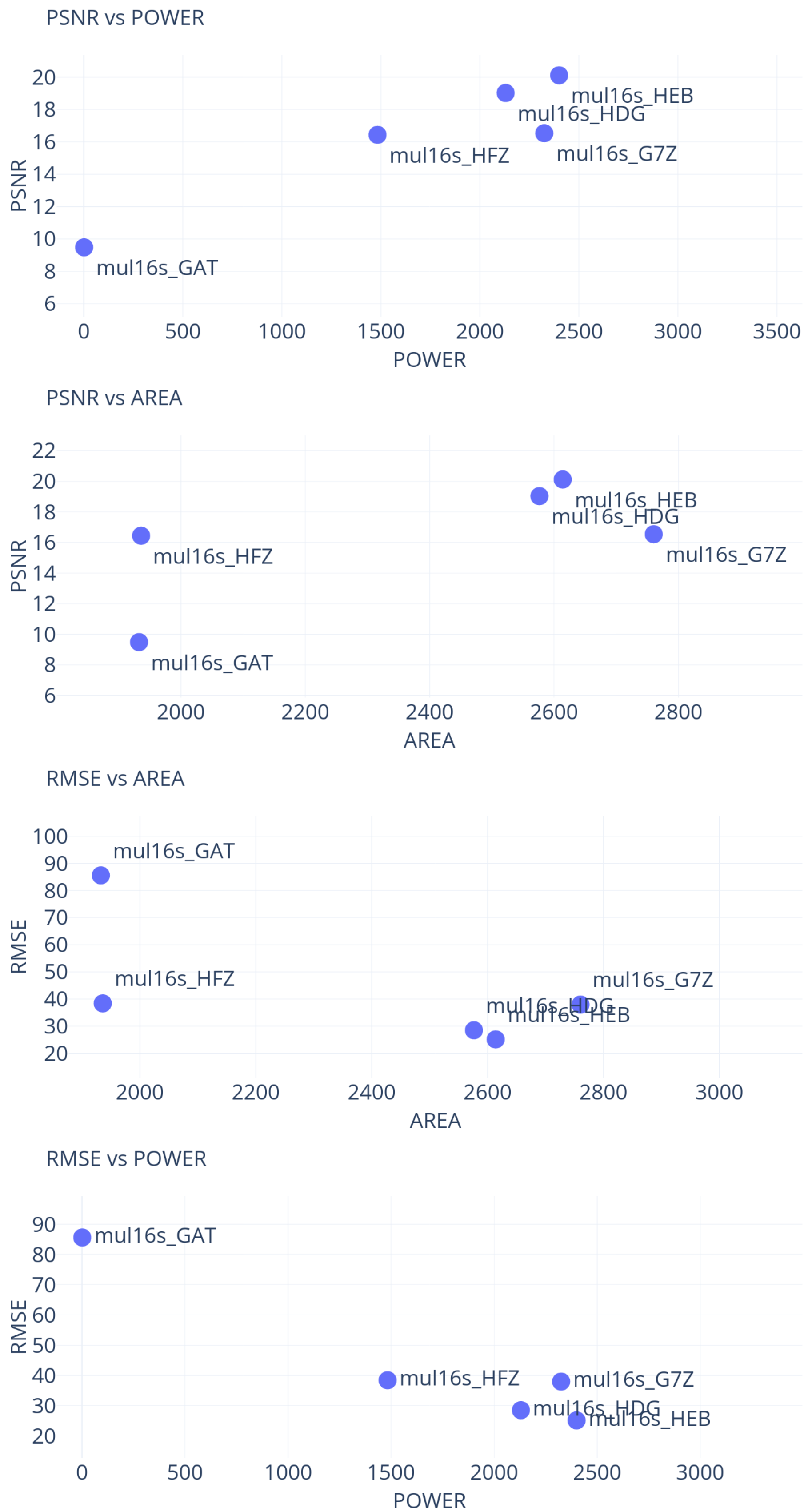


Figure 7.2: The image quality (PSNR,RMSE) obtained by applying various 16-bit approximate multipliers from EvoApproxLib. Power and Area are given for a single multiplier in μm^2 unit of measurement.

Chapter 8

Approximation of a multiplier

The ultimate goal of this thesis is to design and implement a method to automatically perform approximations of selected arithmetic operations in the chosen image filter. In particular, we will deal with the design and evaluation of an approximate multiplier for the NL-means image filter. The more specific objective here is to find a balance between the quality of filtering and the potential implementation cost (area, power consumption) of the approximate multiplier used in our target image filter. In other words, ideal result is to find as biggest reductions as possible, but the difference between original filtration and approximated filtration should not be almost noticeable by independent an viewer.

So far in this thesis we discussed a few options how to approximate arithmetic operations. To summarize our objective:

- We will need floating point to fixed point conversion because the original implementation is coded in floating point and the target multiplier has to be signed and work in the fixed point arithmetic.
- As EvoApproxLib does not currently contain a suitable approximate multiplier, we have to develop a new one.

8.1 Specification of the approximation

In order to create an approximate multiplier, one usually start with an exact implementation and performs the so-called functional approximation. This can be done manually or algorithmically. We will follow the second approach and use CGP to create approximate multipliers.

Approximating a multiplier is a multi-objective optimization problem where the error and non-functional circuit parameters are conflicting design objectives [2]. We have totally three attributes to look upon, Area on a chip, power consumption of a circuit and Error measured by suitable metrics. From these three, it is the best practise to choose one, which will have the most significant impact and the approximation will be built around this parameter. For example, we can divide this multi-objective problem into more single-objective optimization. Let us say we choose the area of a chip as main objective. Then we can approximate a circuit until this criteria is met (e.g 80% size of original circuit). After reaching this constraint (e.g., the circuit size is not below 80%) we can start new approximation trying to lower the error as much as possible. Of course this approach and

prioritization is only a expiric and we cant predict which setting will bring out the best results, thus finding a way through this division will be part of experiments on this thesis.

Another question arises how to evaluate the partial results or even the finale result of approximation. A crucial question is then how the error of a given approximation is derived [2]. Best fitting error metric for this particular use case has to be chosen and also experiments with various metrics will be carried out. But probably only two choices are viable in this type of approximation.

- *worst case absolute error*. In systems that are time based or are dependable, this type of error is very important. Even in image processing in some cases worst case error could have a big impact on resulting quality of filtering.
- *mean absolute error* on the other hand is a common choice in these applications as it naturally reflects the approximation objective in terms of error.

8.2 Determining bit width for a new multiplier

In the beginning of this approximation we need to decide about key parameters of the approximate multiplier. First matter to sort out is finding the ideal bit width of multipliers used in this application. For answering this problem we might be able to use information specifically from section 6.4 and graph 6.11.

From the experiments we know that for integer part we need at least 10 bits. It also determined that using more than 10 or 11 bits for IWL is not beneficial because it has no impact on quality of a result. With this knowledge we are safe to set value IWl to 10 bits.

Now we need to determine the value of fractional word length. From the graph 6.11 the best compromise is using 9 bits for fractional part.

Finally we can now set the ideal bit width of new approximate multiplier. 1 signed bit, 10 integer part bits and 9 fractional part bits, in total 20 bits multiplier. In comparison with the original implementation we already save 12 bits even before approximation.

8.3 Approximation method

The approximation starts with a fully functional multiplier and parameters. There are two approaches how to evaluate the quality of a candidate approximate multiplier. The first one uses statistical information about what values are floating into multiplication operations, e.g histogram 6.6. From this histogram we can create a discrete distribution from which we will randomly generate pseudo-random set of input numbers to evaluate each candidate solution. Each candidate filter will be evaluated using MAE against an exact multiplier and considering the generated input vectors.

The second approach, a candidate multiplier is directly used in the image filter and the image filter is employed to filter a training noised image. The fitness score is then established as RMSE or PSNR with respect to a perfect image.

Both these approaches has pros and cons. More comparison information between these two procedures are in subsection 8.3.2.

8.3.1 Conversion of the circuit to CGP chromosome

An exact 20 bit multiplier was modelled using the * operator in Verilog and synthesized into a netlist. The multiplier was encoded in Verilog file and the functional blocks were

represented as 2 input gate. As described in section 4.1, CGP uses a chromosome for its structural representation which is an array of integers. In CGP, any combinational circuit can be represented as an 1D structure as a list of used gates. Hence, we set:

- M (number of rows) = 1, N (number of columns) = number of gates. In this particular case, Verilog file of the multiplier contains 118

  wire [39:0] _xzcv_,

  and 98

  wire _xzcv_,

  of these type of gates. This means that the total number of gates will be $N = (40 * 118) + 98, N = 4,818$ gates/functional blocks.

- As the result of this structuring the L-back parameter had to be set to its maximum value.

8.3.2 Error computation based on simulation

After obtaining a CGP chromosome representing a candidate multiplier its error has to be computed. The easiest solution is simulate the bit flow through CGP function. However this approach is not scalable and practically does not work for 12 and more bits [12].

Due to that, the error is commonly estimated using a subset of input vectors only, e.g. 10^8 inputs were used to evaluate 16-bit adders in [7]. In this thesis we will use knowledge of inputs histogram representation to create a better statistical tool for input vector set generation. Of course, for obtaining possibly better results we could evaluate not the candidate multiplier, but the filter in which the multiplier is employed. This option will not be examined for one good reason. One walk through of the filter implementation consist of tens of millions multiplications even on the smallest images. using the exact simulation with this approach is too much time consuming and it would take days to even simulate few runs of the filter.

8.4 Error metrics definition

As described in section 8.1 possibly the best metrics that can be used in this particular example are *worst case average error* (WCAE) and *mean absolute error* (MAE).

As said before approximation starts with exact 20-bit multiplier. We will call this circuit G, a golden circuit. Furthermore examined circuit which is an approximated version of golden circuit will by called C, a candidate circuit. These circuit candidates calculate two functions f_c resp. f_g, where:

$$f_g, f_c : \{0,1\}^n \longrightarrow \{0,1\}^m \tag{8.1}$$

We can now slightly modify metrics described in chapter 2.2 in order to reflect the fact that the error will be evaluated using a set S of vectors sampled from the normal random distribution with parameters of this distribution, so it would be as close as possible to the operands values from the histogram 6.6.

$$\text{WCAE}(G, C, S) = \frac{\max_{x \in S} |\text{int}(f_g(x)) - int(f_c(x))|}{1} \tag{8.2}$$

$$\text{MAE}(G, C, S) = \frac{\sum_{x \in S} |\text{int}(f_g(x)) - int(f_c(x))|}{|S|} \tag{8.3}$$

where int(x) is representing a integer value of bit vector x [2].

8.5 Design of the approximation process

The proposed CGP-based approximation is presented in this section. Chromosome is divided into three main parts: input (as representing two 20 bit input numbers), circuit content (2247 gates), output (40 bit output number). The circuit is also divided into nodes, each of them is composed by three values. Two denotes its input and the third a function. All types of logical gates used in the implementation can be seen in table 8.1.

Gate	IDENTITY	AND	OR	XOR	INV	NAND	NOR	XNOR
Size	0	2.34	2.34	4.69	1.48	1.87	2.34	4.69

Table 8.1: Types of logical gates and their relative size in hardware implementation according to [12].

CGP starts with one parent chromosome, representing an exact 20 bit signed multiplier. By applying the mutation operator we discover new chromosomes and create new generations. We will investigate two options for mutation operator in this thesis:

- Classic random mutation.
- Probability map mutation proposed in chapter 9

Every chromosome is tested minimally on two parameters combining $(area, power) \times (MAE, PSNR, RMSE)$. This pair is then used in the fitness function. Evolution continues until a termination criteria is met. in our case it is normally set on the maximum number of generations.

Optimization problem is: Based on initial circuit G and a threshold τ it is our target to find approximated circuit C, which has a minimal size and satisfies a formula $MAE(G, C, S) < \tau$.

Accordingly to this problem we can set up our exact fitness function used in this implementation:

$$f(C) = \begin{cases} size(C) & \text{if } MAE(G, C, S) < \tau \\ \infty & \text{otherwise} \end{cases} \tag{8.4}$$

The circuit simulation is the most time consuming part of this approximation process. It is only logical to run it as frequently as possible. Thus algorithm is designed to test an approximated circuit on its size first and only new smaller models are passed for the error evaluation. this way we dont waste computation time on circuits that do not meet our requirements. Full flow diagram can be seen in figure 8.1.

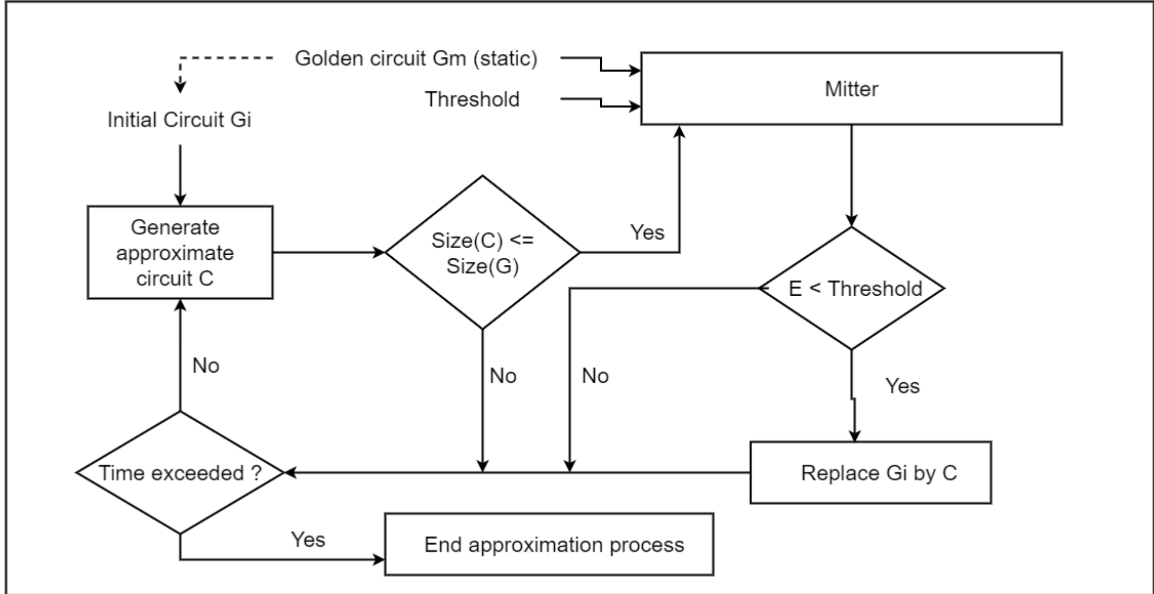


Figure 8.1: Flow diagram of the whole approximation process.

The approximation process is based on CGP and its $(\lambda + 1)$ search strategy as seen in Alg. 3.

```
Result: Approximated 20 bit signed Multiplier
G = Golden_circuit;
Best_circuit = Golden_circuit;
C = Golden_circuit;
Gen_arr[number_of_offsprings] = Mutation(G);
while Time limit exceeded do
    forall C = Gen_arr[i] do
        if Size(C) <= Size(G) then
            if err_threshold > MAE(C,G,S) then
                Best_circuit = C;
            end
        end
    end
    G = Best_circuit;
    Gen_arr[number_of_offsprings] = Mutation(G);
end
```

Algorithm 3: algorithm showing CGP evolution process.

8.6 Using parallel simulation for better performance.

As said before, experiment showed, that exact simulation is viable only for a set of input vectors with size up to approx 2^{12}. Hence this approach is intractable for our 20-bit multiplier.

There is a way how to accelerate this method with the so called parallel simulation. We simply do not compute with bits but with bit-wise arithmetic. We use the fact that,

for example, in the C language there are bit-wise logic operators. Then it depends on a particular processor architecture to what extend we can speed up the process.

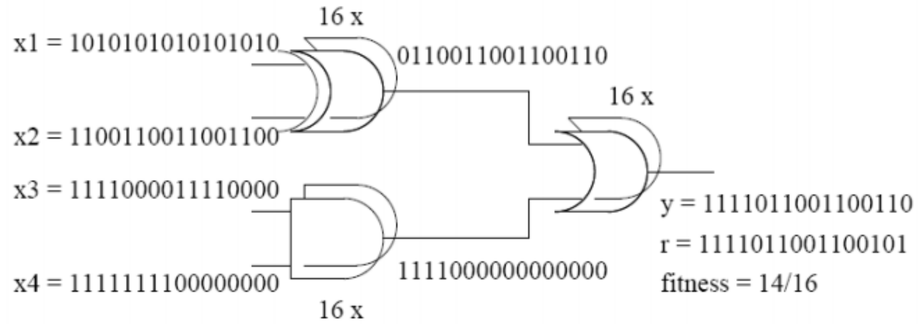


Figure 8.2: Example of parallel simulation of a random circuit.

For example, in figure 8.2, we have a circuit containing 4 inputs, 3 logical gates, 1 output and realizing a function $y = h(x1, x2, x3, x4)$. Next we assume that the goal of evolution is to find a target vector r representation for all possible inputs. In naive simulation we would apply 2^4 input vectors one by one to the circuit. But if we can store in this example 16 bit vector to the integer type variable in C language, we can feed the circuit with not bits, but whole vectors and apply logical operation on integer numbers representing 16 inputs. So in only one simulation run we can gain result vector which we compare with desired outcome and compute fitness function. Of course nowadays current processors supports 64 bit or even more complex operations. This means that we can speed the simulation time up to 64 times. In general k times on k bit processor. The parallel simulation was applied to evaluate candidate 20 bit multipliers using a test set S sampled from the adjusted normal probability distribution.

Chapter 9

Acceleration of CGP with a new mutation operator

Mutation is a vital operation in the whole evolutionary process. It can influence both exploration and exploitation of a state space. Its settings significantly influences the quality of evolution. There are two basic approaches to mutation operator used in CGP.

- First, we set a target probability value x. Then on every gene of the chromosome we decide with probability x if the individual will be mutated.
- Second, We set a constant value x. Then we will randomly choose x genes that will undergo the mutation.

Of course these approaches are suitable when the output of CGP is not weighted. It means, in case of a binary output, that every bit has the same impact on the quality of a result (e.g. parity circuits, decoders). On the other hand, with multipliers and arithmetic circuits in general, the bit position matters. There is a big different, if you have correct bit value in LSB (least significant bit) or in MSB (most significant bit).

- Error in LSB gives us Error equal to $-1, 1$ in terms of integer data types.
- Error in MSB rockets the error value to possible 2^{n-1} values for a bit vector with size n depending on the number data type representation.

Applying the standard mutation operators to a weighted output in this particular use-case seemed to be very inefficient. First There is a big chance that it could lead evolution to the wrong direction in many cases. There would be many candidates with huge fluctuations in terms of quality of the output. Second the exploration effect would be much bigger than exploitation and in this case it is most likely unwanted. Third, there is uncertainty that it would lead evolution to viable result and it would consume too much process time. This led to a creation of a new concept of a mutation operator which we will call The probabilistic map heuristic (PMM).

9.1 Principles of the probabilistic map heuristic

In this thesis we propose a new principle called probabilistic map. This method should solve or improve some evolutionary shortcomings in some specific tasks by changing the mutation strategy.

The whole idea behind directing the evolutionary process is, as said in section above, that every bit in multiplication output has different weight. If our goal is to decrease the size or power consumption by introducing a specific error to a multiplier, we want the error to be as small as possible. In order to achieve that logically, we do not really care about error margins in bit positions near The LSB bit. On the other hand we do really care about good precision in bits positions around The MSB bit.

From CGP structure we know that to obtain the value of a certain bit position on the output we need to follow a path of calculation through function blocks. These block calculations have a direct impact on the result value. Every output bit has its own path and the general idea is to assign specific probabilistic value of mutation for every function block. Function blocks that are on a path to the LSB bit, will have a bigger probability of a mutation and function blocks to MSB will have a very small chance to mutate.

The basic idea is to lover the multiplier attributes like the size or power consumption in function block on the paths, that does not have much influence or the overall quality of a result. To achieve this, we use a principle of a very well known learning heuristic in neural networks - Back-propagation.

9.1.1 Basic Principle of Back-propagation

This method is the most frequently used learning approach for neural network, but we use its basic ideas for setting up an algorithm that creates a probabilistic map. The algorithm of back propagation starts when the neural network finishes its forward calculations and gives us the results. The error is then computed according to a loss function. Back propagation takes this rate of error a propagate it back through a neural network. Every error value is propagated back through a path, that had a direct impact on the scale of an error. With this information neurons along a certain path adjust their weights values according to a error magnitude so it can produce better result for next learning epoch.

Propagation itself is calculated with partial derivations as seen in Fig. 9.1. We basically want to calculate difference Δw_j. For finding out the partial derivation we need to use the chain rule shown again in Fig. 9.1:

$$\frac{E_{total}}{\delta w_j} = \frac{E_{total}}{\delta out_i} \cdot \frac{\delta out_i}{\delta in_i} \cdot \frac{\delta in_i}{\delta w_j}. \tag{9.1}$$

If we substitute the general chain rule by values in Fig. 9.1 we get specific partial derivation:

$$-\frac{wanted}{out_i} \cdot (out_i \cdot (1 - out_i)) \cdot out_j \tag{9.2}$$

And this way neural network uses error margins to learn. In creating a probabilistic map we use a little bit different type of back propagation. We will not calculate exact error and then propagate it back. But we will use the fact that we can distinguish importance of each bit position in the output vector. Based on that we propagate back the information of how much has a given path influences the total error.

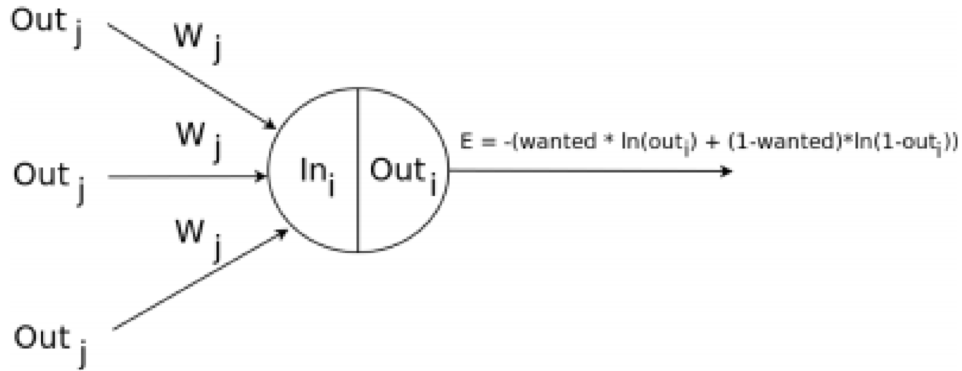


Figure 9.1: Neuron in the last layer in a neural network [2].

9.2 Algorithm for creating the probabilistic map

In this particular use case, the CGP has outputs vector containing 40 bits. In addition we have 40 paths to calculate for back propagation algorithm. But first we need to decide probability values for each output bit path. In testing cases, probability $p_0 = 0.001$ was set to a path to the LSB bit and probability $p_{39} = 0.0000001$ was set to a path to the MSB bit. Bits between are set linearly with respect of bit position in the range $(0.001, 0.0000001)$. For each CGP function block, this probability is calculated according to a block position on the path. If a function block is on multiple paths, The highest probability out of all relevant probabilities is stored.

Of course this decision is questionable. Why would higher probability value had a bigger priority. Logically it is not ideal because we rather do not want to change more significant bit values so smaller priority should have a higher priority. And it is true. But in this use case we have a specific type of CGP. Specific in the way that we use CGP with rows count set to 1 and the number of columns is 2247. This particular CGP chromosome is only one dimensional sequence of functional block. As a consequence, paths of a LSB bits are much shorter than a paths to the MSB. More precisely the path to the LSB bit usually contains about 2 or 3 functional blocks but MSB bit path contains 1500+ functional blocks. As a result of this settings if we choose the second variant almost all functional blocks will have probability of mutation 0.0000001, and that is not good. If we choose first variant of preference the probability values spread evenly and it has the keen effect. But if a CGP has more than 1 row, second variant could be more preferable.

after evaluating the probabilities along with the paths all used function blocks have a probability number for mutation. Of course one can mutate input of a function block or its function as well. The probability map is calculated for every chromosome before mutation

in every population. The initial probabilistic values for bit position and its path does not change during the evolution.

```
Result: Probability_map
Prob_array = Manually enter probabilities for every bit position in the output;
forall Bits in the output in position i starting from LSB do
    Next_gate_block_arr = chromosome(i);
    for (k = cgp_columns; k >= cgp_input_size * 2; k − −) do
        if k is in Next_gate_block_arr then
            input1,input2 = get_inputs_from_current_gate(k);
            Next_gate_block_arr += (input1,input2);
            if Probability_map[k] is not set then
                Probability_map[k] = Prob_array[i];
            end
            Next_gate_block_arr -= k;
        end
    end
    clear Next_gate_block_arr;
end
```

Algorithm 4: The probability map for mutation.

Chapter 10

Results of the approximation

In this chapter the results of the approximation process described in section 8. Since the goal is to find best trade-off between error and implementation cost, we will be comparing multiple approximated multipliers under different error metrics values as the main approximation objective. The second objective is to evaluate the proposed mutation operator based on the probabilistic map.

10.1 Approximation and program arguments

At first, it is important to go through both program and approximation process arguments. The settings of the arguments have a tremendous impact on the quality of a evolution. The program is written in C/C++.

- num_of_colums: As it was mentioned in 8 The CGP created from Verilog file of the exact 20-bit multiplier have a specific format. It has only one row and basically the number of functional block is equal to the number of columns. In this specific example in every simulation we use value 2247 as number of columns. The exact program is programmed generally so it can handle reasonable CGP setup.
- num_of_rows: is set by default to one.
- component_input and component_output: this arguments describes the number of primarz inputs and outputs. Specifically we approximated 20 bit multiplier, thus logically input is every time of size 40 and the output is 40.
- block_input: this parameter determines the number of block inputs. We are only using binary and unary logical operators. since the majority of operation used are binary this value is set to 2.
- block_count: The number of functions in the set of CGP. see Table 8.1 for the used setup.

Now the arguments that have influence on evolution.

- population_count: in general this value specifies an end condition of a approximation process. In this thesis we use combination of two ending conditions. The first is number of maximum generations allowed. The second is time condition. The reason behind this combination is the experiments showed it would be probably best practise

since both program process time and number of generations have great influence on quality a the results.

- The number of generations. In the beginning of the approximation almost all individuals. satisfies the area or power constraint and, hence all of them are evaluated in terms of error. Based on preliminary experiments, default number of generations is 1500.

- The l-back parameter sets the connectivity in a cgp. In this case we do not have really choice of setting it to other values than maximum one (total connectivity). This ensures that the arbitrary combinational circuit can be created on a given number of gates.

- Threshold parameter contains a error metric value. Specifically we use MAE as approximation objective. The MAE metric is not relative, thus it is not represented in percentage but in this case integer value. Since we are transforming floating numbers before multiplication to fixed point arithmetic, we are working in integer values level. So far only the floating point number histogram of the image filter was shown in Fig. 6.6. But if we look at the integer representation of the histogram we get a slightly different result. the range of integer numbers is $< -595398, 595398 >$ and it has the same shape as normal distribution. The experiments performed in this thesis uses the threshold values 500, 1000 or 10000. So this is relatively small error in comparison with the actual 20 bit multiplication range.

- The last parameter is the mutation probability. First we will discuss the common mutation of CGP. As said before in this chapter two concepts of mutation will be compared. One thing they have in common is how the settings affects the whole evolution. Experiments showed that in the mutation chance is too high, for example, it is set to 0,1 or 0,01, then the approximation always leads to unsatisfactory results. Mainly because it converges too quickly. if one introduce to a chromosome too many changes at once, it increases the chances of making bad mutations in single generation. Then it is too easy for evolution process step out from right direction. And as it was described in chapter 9 if an error occurs in bits around LSB it has much smaller impact then error near MSB bits.

  On the other hand if you choose a very small chance for mutation, it will take a large amount of time to even converge somewhere. And it would take forever for the program to come to a good solution in reasonable time.Base on our preliminary experiments, best value for compromise between convergence and program duration was mutation probability set to 0,0001 in common mutation operator.

  To avoid this compromise and get hopefully better results the concept of probability map was proposed in chapter 9. Probability of mutation on the LSB path is set to $0,001$ and probability on the MSB path is set to $0,000000000001$. Probabilities of the remaining paths (and so functional blocks) depends on the bit position pos and equal to $10^{-1 \cdot \frac{pos}{2}} * 0.001$.

10.2 Results for 20-bit multiplier using the probability map mutation

Using a Salomon supercomputer in Ostrava gave me a tool to run 24 instances of the approximation program. We used the following CGP parameters: $col = 2247, rows = 1, c_in = 40, c_out = 40, pop_size = 50, block_i = 2, l_back = 2287, funcl_b = 8, threshold = 500, max_time = 48hours, max_gen = 1500$. The set of number S contains 128000 pseudo-random generated numbers for every candidate multiplier.

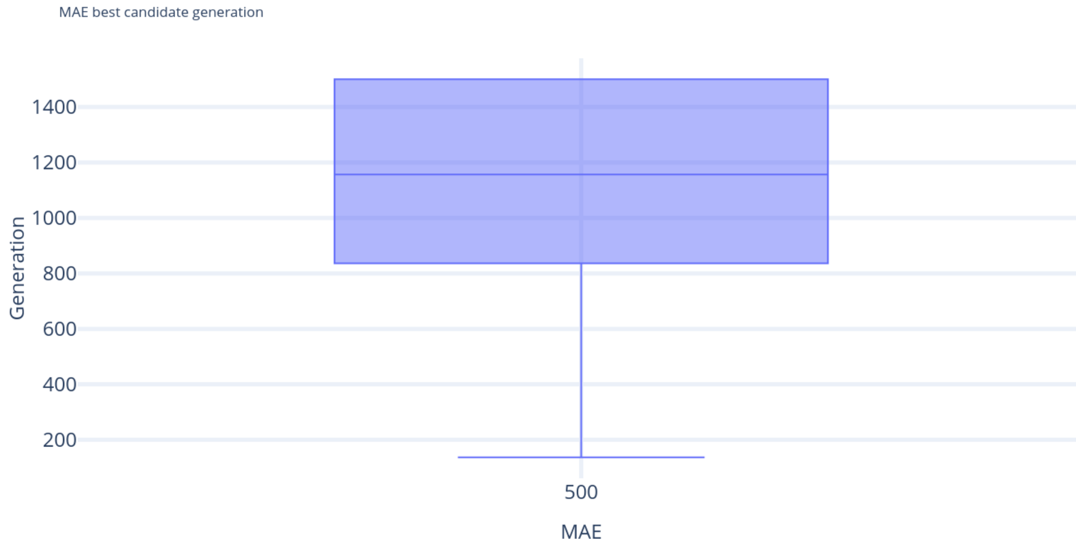


Figure 10.1: Number of generations needed to find an approximate multiplier with MAE = 500. (Constructed from 24 independent runs with the probabilistic map mutation, max. 1500 generations.)

The boxplot in 10.1 shows the number of generations needed to find a n approximate multiplier with MAE = 500 Constructed from 24 independent runs. The exact statistical numbers can be found in table 10.1.

best	worst	average
$6739\mu m^2$	$6942\mu m^2$	$6825.25\mu m^2$

Table 10.1: Average, best and worst area value obtained by Prob Map mutation.

10.2.1 Examining the evolution process

Fig. 10.2 shows the average area reached in the the course of evolution calculated from all 24 CGP runs. From that graph we can deduce some information how probabilistic map influences the evolution.

First obvious thing is that the result is converging slowly but steadily. This is a good result because we do not want to end up quick in local optimum. It also means that settings of the program are balanced and it has good impact on the quality of a result.

Also it tells us that maybe if we let the process evolve longer we could get a little bit better results.

Also in graph 10.3 we can see progress of the run with best found result. It only confirms all findings from the previous plot. Another interesting observation is that the progress is not linear as it would seem in the average graph. It can be seen that in first 400 generations we can see the best performance in sense of progress.

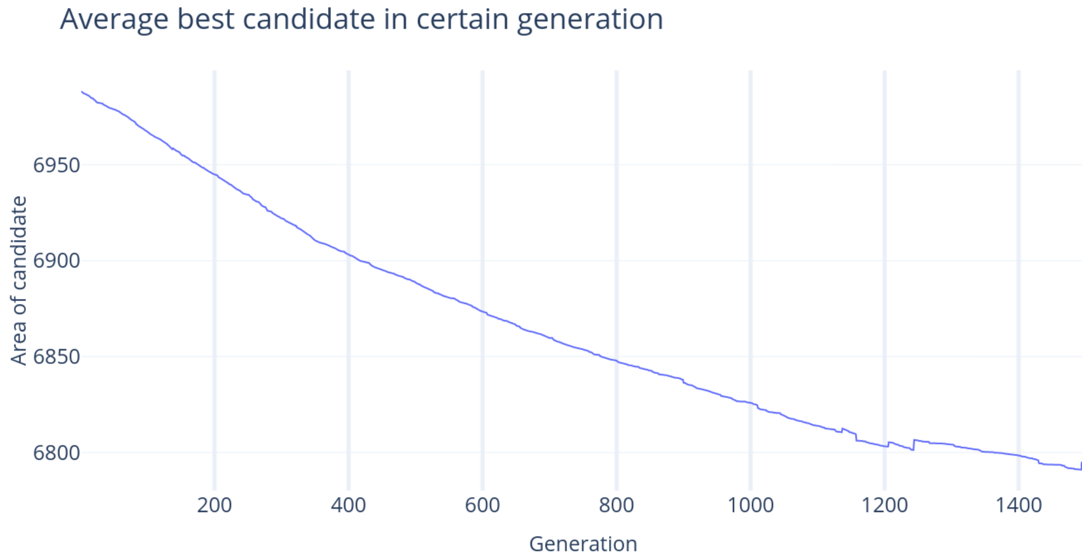


Figure 10.2: The average best size of the approximate multipliers during the evolution (probabilistic map mutation).

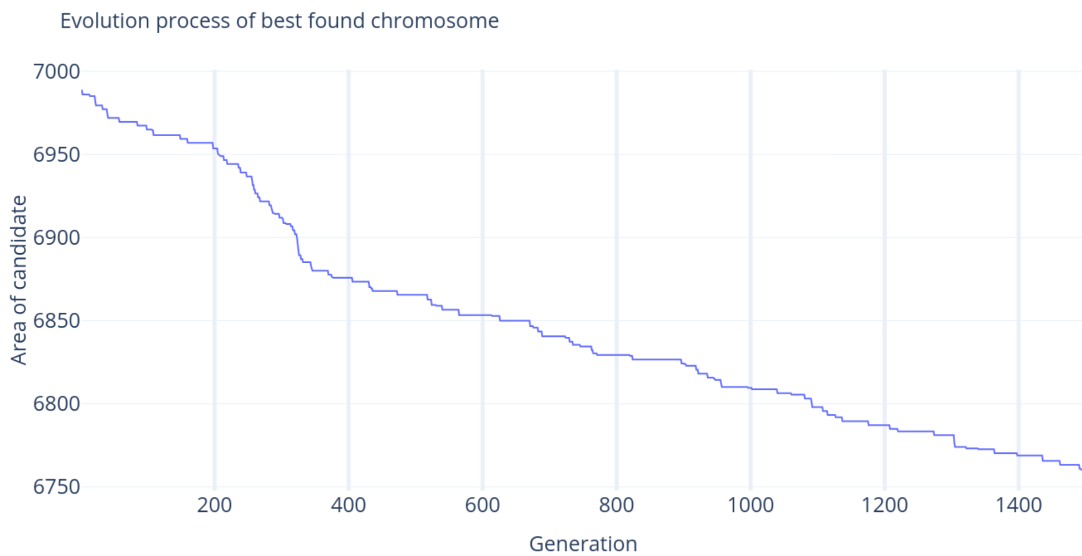


Figure 10.3: The area of the best approximate multiplier during the evolution (probabilistic map mutation).

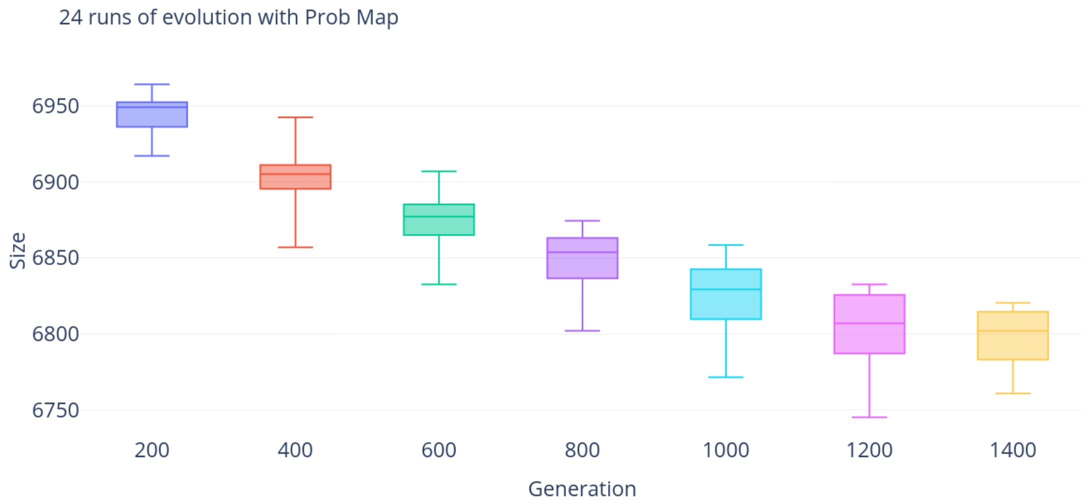


Figure 10.4: Boxplot showing candidates and their size in different generations. (probabilistic map mutation).

We can also compare evolution process between two different MAE threshold settings. in graph 10.5 We can see that evolution progress of both approaches is more or less similar. The speed of the convergence is comparable. Of course two similar settings will never have identical progress. The experiments shown that time complexity for calculating the probability map is negligibly small in caparison with other parts of implementation e.g. vector based simulation and so on.

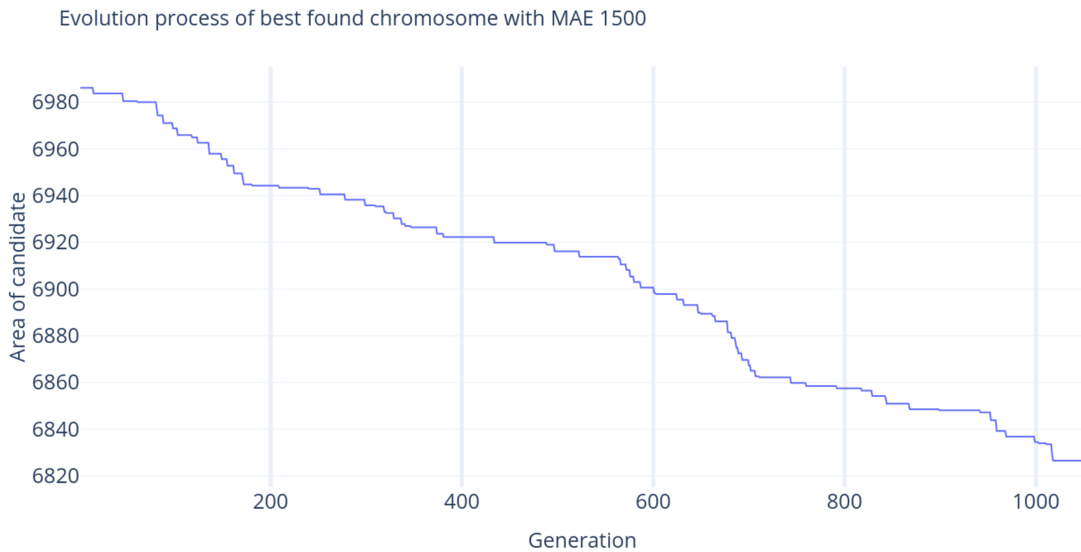


Figure 10.5: Average best size of a candidates in certain generation of the evolution (probabilistic map mutation).

10.3 Results for 20-bit multiplier using classic mutation

In this section, the common mutation of CGP will be compared with the Probabilistic map. Mutation rate was set to value 0.00001 so that chance for mutation is rather small. But additional experiment confirmed that this is the best settings we tested fo common mutation. If the mutation value is higher, e.g. 0.001, majority of runs ended up with the evolution in the very early stages. It means that most of the candidates among first generations were mutated so badly that they didn't satisfied the MAE threshold even they had almost the same size as unapproximated multiplier. If the mutation number is too small chromosomes were staggering on the same size for hundreds of generations.

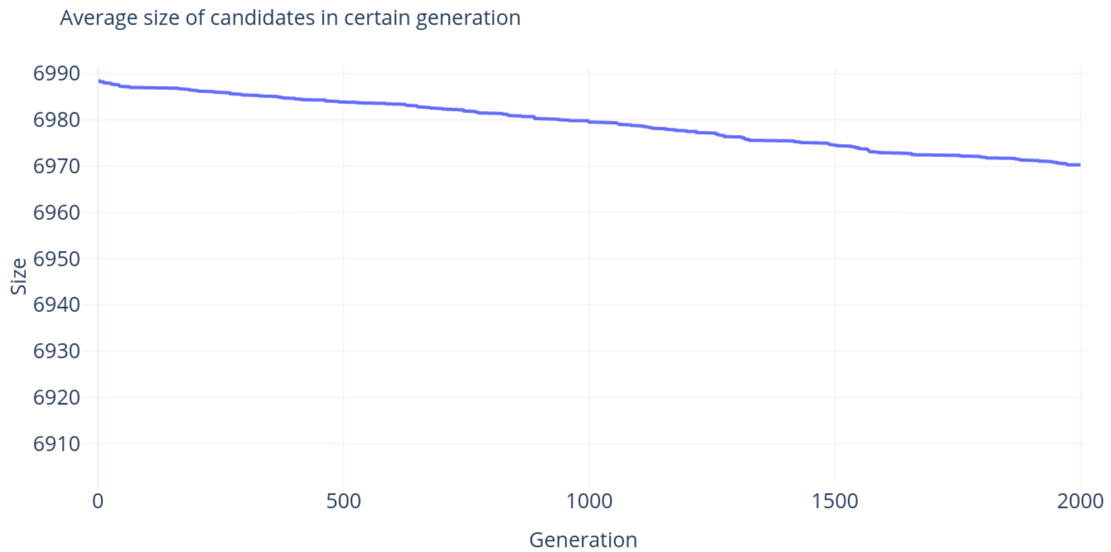


Figure 10.6: Average size of the best approximate multiplier during evolution (common mutation).

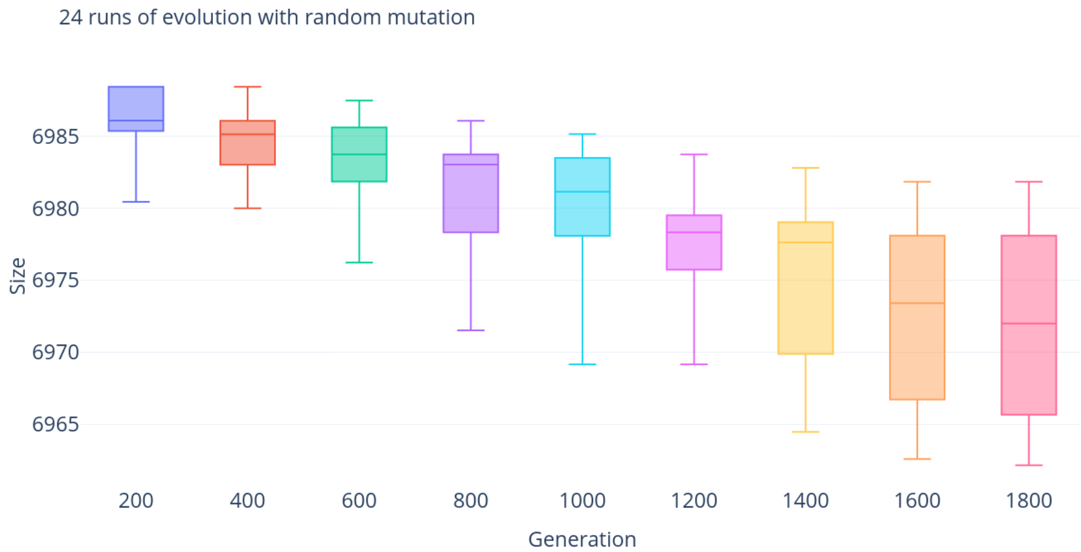


Figure 10.7: Boxplot showing candidates and their size in different generations (common mutation).

Fig. 10.6 shows the average size of approximate multipliers when common mutation is used. If we compare this plot with plot representing probabilistic map 10.2 the convergence is much smaller and even the results are much worse. This only confirms the assumption that probabilistic map would contribute to the faster convergence.

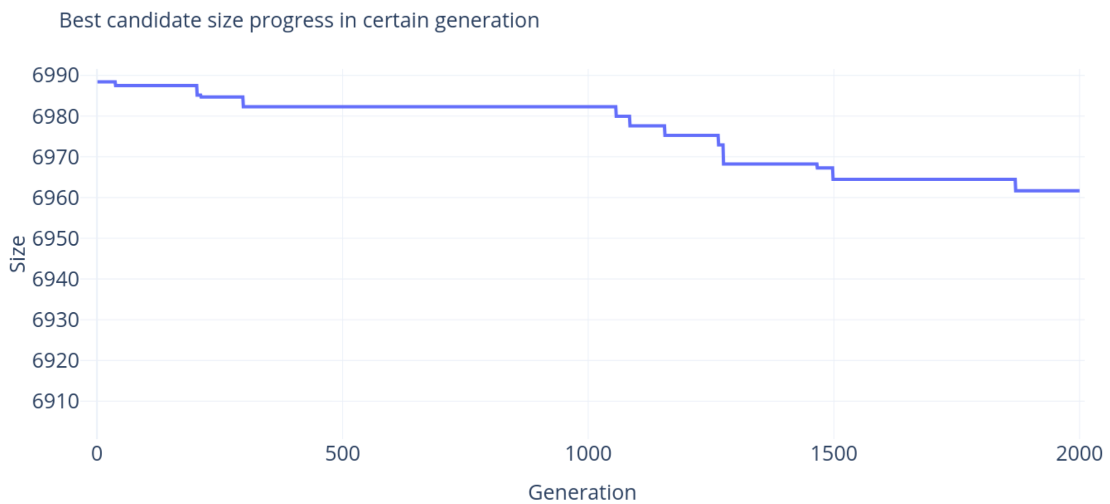


Figure 10.8: The area of the best approximate multiplier during the evolution (common mutation).

If we compare the results of best candidates, we can see that the candidate which was evolved using the probabilistic map, converges much quicker an also in much smaller steps. This also confirms our assumption about the probability map based mutation. It is logical if we let bits around LSB mutate more quickly than those around the MSB, better solutions are then obtained.

If we compare value from tables 10.2 and 10.1 we will see that there is a difference between two best found chromosomes about 222 μm^2 which is a lot, The best chromosome obtained using the common mutation achieved MAE value of 470 and Probabilistic map chromosome of 375. The random mutation chromosome was found in generation number 1869 and Probabilistic map chromosome in 1057. In conclusion, the approximate multiplier obtained using the probabilistic map based mutation is more compact, produces smaller error and was found in fewer generations.

mutation	best	worst	average
common	$6961\mu m^2$	$6984\mu m^2$	$6971.18\mu m^2$
probability map	$6739\mu m^2$	$6942\mu m^2$	$6825.25\mu m^2$

Table 10.2: The average, best and worst area of approximate multipliers obtained by using both mutation approaches comparison.

To point out the conclusions drafted in previous paragraph, we can look at plots 10.9 and 10.10. First graph is showing boxplot with MAE values found by all CGP runs. It can be seen that both approaches provide similar results in terms of MAE, but the probability map based mutation leads to more stable results.

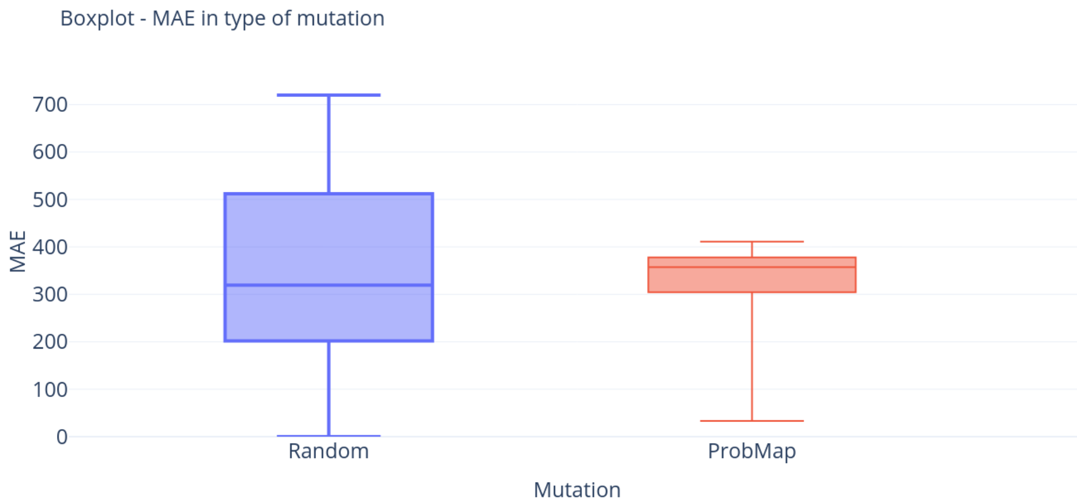


Figure 10.9: Comparison of MAE values obtained using the proposed mutation operators.

In that case of the size of approximate circuits, PMM clearly leads to better results. As it can be seen even the best approximated multiplier using common mutation is bigger than the worst candidate from probabilistic map.

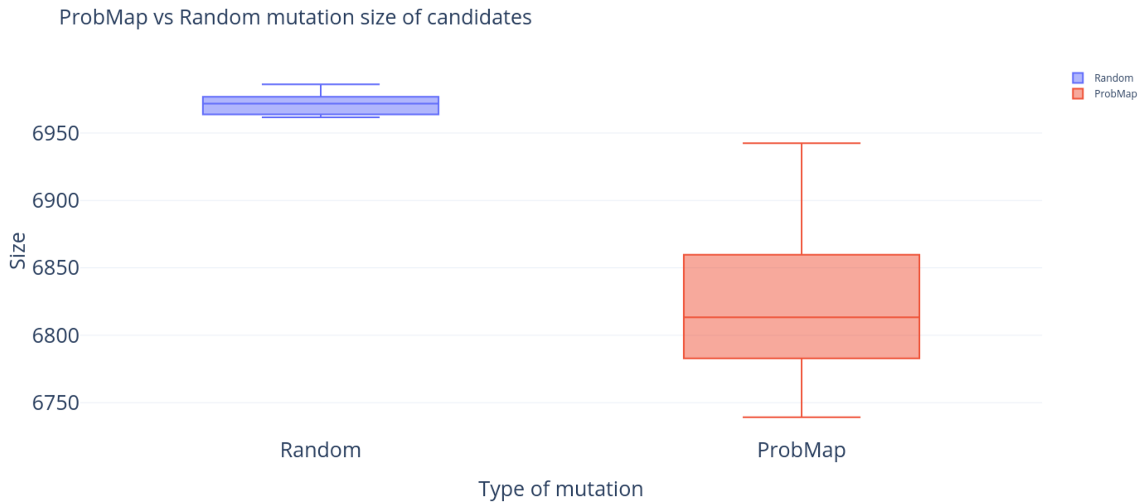


Figure 10.10: Comparison of sizes of approximate multipliers obtained using the proposed mutation operators.

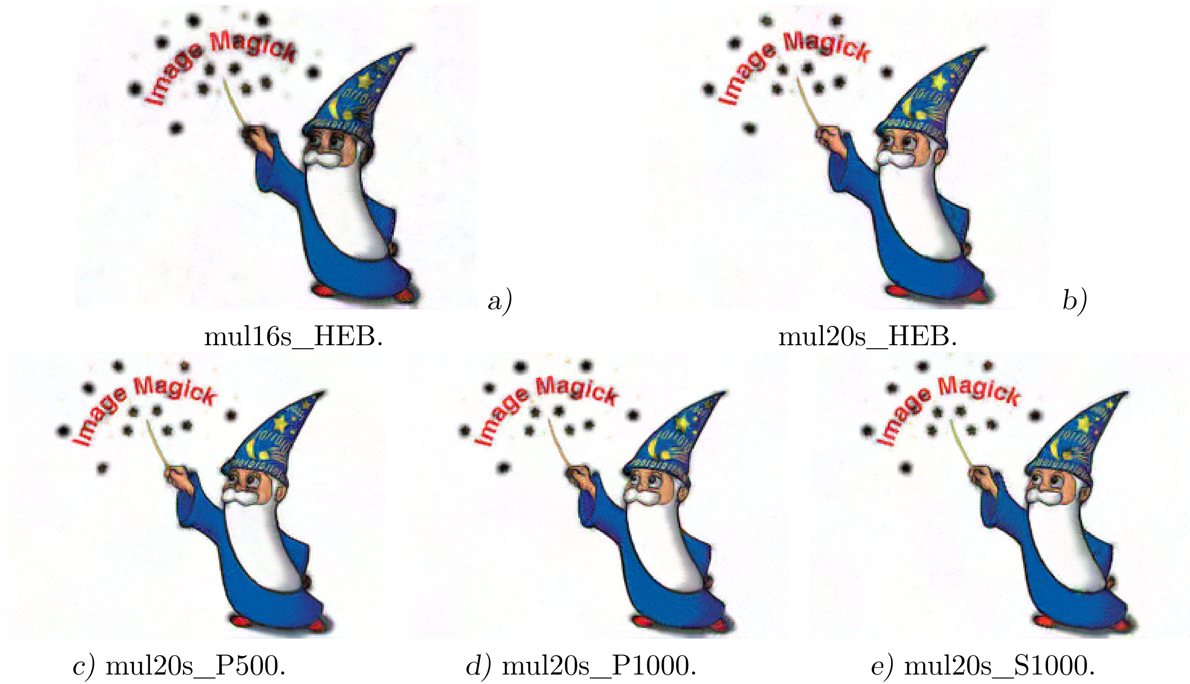
On the other hand there are some issues that need to be addressed. Due to the enormous time complexity of the algorithm it was impractical to test the evolution on more than 2000 generations. For this number it took the program more or less 24 hours to finish (on used CPUs) even with many optimization techniques such as parallel simulation and using optimization compiling C++ flags and so on.

From the behaviour we can see that the random mutation is progressing slowly but surely. Nearly 3/4 of evolution runs with random mutation didn't end their search before the generation 2000 and would continue to find possibly better solutions. In the case of Probabilistic map it was around 1/4 of the candidates.

10.4 Applying approximated multipliers with image filter to image dataset

In this section some of the best chromosomes found by evolution will be chosen and used in the image filter on various images to see results. Of course the resulting multipliers will be tested on the same image as every other technique described in this thesis for the best comparison effect. We will report the PSNR and RMSE.

If we look at the set of pictures 10.11 we can see how the chosen unapproximated and approximated multipliers used in the non local denoising image filter affects the image quality. Two of the multipliers are unapproximated. The first mul16s_HEB is a 16 bit multiplier is shown fo comparison with the 20 bit multipliers and the mul20s_HEB is the 20 bit unaproximated signed multiplier. Then we have mul20s_P500 which is the best approximate multiplier obtained by probabilistic map with MAE threshold under or equal 500. The same goes for mul20s_P1000 which is also generated with probabilistic map-based CGP but with MAE threshold under or equal 1000. The last one is the only representative of random mutation approach with MAE threshold also under or equal 1000.



a) mul16s_HEB. *b)* mul20s_HEB.

c) mul20s_P500. *d)* mul20s_P1000. *e)* mul20s_S1000.

Figure 10.11: Results of applying different types of approximated multipliers to image filter implementation.

On the first sight there is not much visual difference. And with that we saved $284\mu m^2$ size on the chip area. On the hand the 4 bit difference between 16 bit and 20 bit architecture is more noticeable mainly on the details.

Table 10.3 is visualizing some statistical data, particularly the error metrics values and sizes of the multipliers. We have to emphasize is that MAE values are not corresponding with PSNR and RMSE numbers. The reason is the way how evolution and fitness function is constructed. Every candidate circuit was tested on exactly 64 thousands input vectors that were sampled from the histogram of operands entering the multiplication in the filter. But the filter contains many other operands that were not modelled in fitness function. Therefore the reason why those two numbers are not matching is simplification of fitness function.

Basically MAE value here represents how the candidate circuit performed in the evolution process. The PSNR and RMSE values shows how it performed in the actual target task, the image filter.

name	MAE	PSNR	RMSE	Size
mul20s_HEB	0	23.75	16.57	$6988\mu m^2$
mul16s_HEB	0	20.12	25.14	$2614\mu m^2$
mul20s_P500	375	23.31	17.42	$6739\mu m^2$
mul20s_P1000	565	23.16	17.72	$6704\mu m^2$
mul20s_S1000	470	23.35	17.34	$6961\mu m^2$

Table 10.3: Error metrics values and sizes for approximate and the exact 20 bit and 16 multipliers.

In Fig. 10.12 and 10.13 we plot the circuit size against the filtering quality. These plots are very similar as MAE and PSNR are highly correlated.

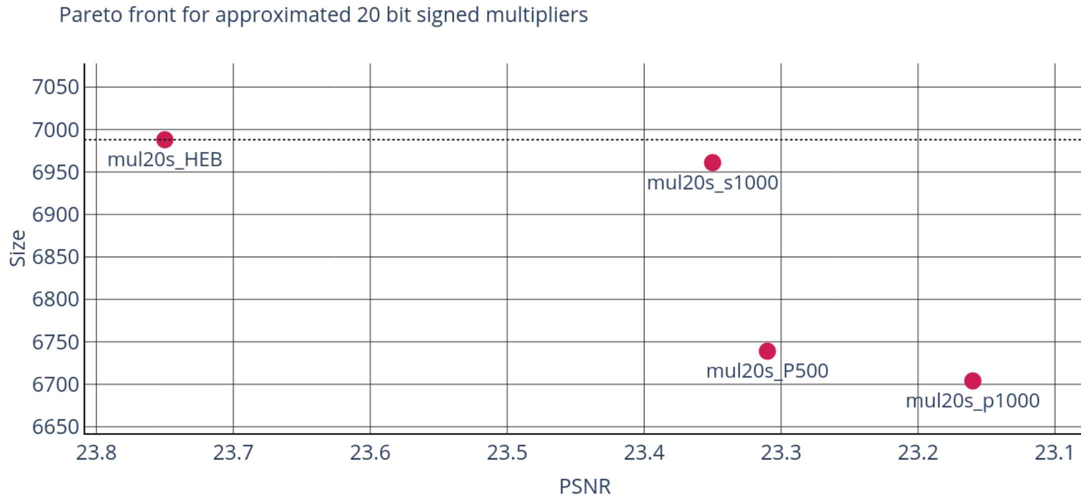


Figure 10.12: The multiplier size and PSNR of corresponding filter on the training image.

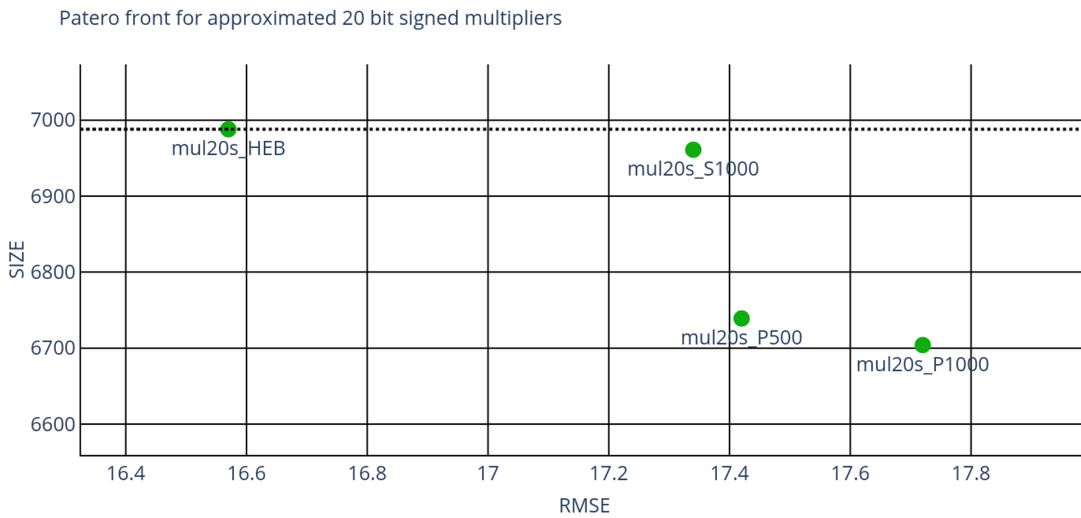


Figure 10.13: The multiplier size and RMSE of corresponding filter on the training image.

10.4.1 Resource Savings

In order to estimate how much resources can be saved by the proposed approximate multipliers, we compared the power consumption and area of a conventional 32 bit FP multiplier and signed 20 bit FX multiplier with our solution. Based on the data provided by Dr. Mrazek, the 32 bit FP multiplier requires approx. 1.48x more area and 1.77x more power consumption than the 20 bit FX multiplier. Our evolved 20 bit approximate multiplier saves 4.06% area compared to the 20 bit standard multiplier. In total, the proposed approximate solution can save 35.28% area with respect to the original 32 bit FP multiplier.

10.5 Real life image denoised by approximated 20-bit multiplier

Noised image with noise parameter $\sigma = 50$

Denoised image with mul20s_p500 multiplier

Chapter 11

Conclusion

This thesis had one main goal and it was to develop an EA based approximation method which exploits particular data distributions in order to approximate an image filter. This goal was divided into smaller parts. First, a suitable image filter had to be chosen as a use case for this thesis. Non-local denoising filter was chosen. After this decision next task was to obtain the operand data distribution of this filter and determine the arithmetic operation which is the best candidate for approximation in the filter. After evaluating that multiplication is the best choice and obtaining a data input histogram the main part of this thesis was to find the best way of the approximation.

We started with reducing the bit width of selected arithmetic operations. We evaluated two basic approaches to converse the floating-point number to the fixed-point representation and then experimented with different bit width settings in this type of number representation. After these experiments we evaluated various approximate multipliers. As a source for these multipliers EvoApproxLib was chosen. Then we experimented with the various 16 bit signed approximate and exact multipliers.

The experiments revealed that 16 bits are not sufficient enough for our case study. As a next step for achieving this thesis goal we decided to create new approximate 20 bit multipliers. Using data distributions and other information obtained from the experiments we modified the evolution process with respect to the chosen NL-means filter. For achieving this task a new mutation operation principle called the probabilistic map was proposed.

We evolved various 20 bit approximate multipliers using the standard CGP and the CGP utilizing the proposed mutation operator. Evolved multipliers were evaluated on the implementation of the NL-means filter. Finally, we compared the quality of filtering of the original and approximate image filter on a set of images.

Results of this thesis shown that approximate computing can be applied in non-trivial image filter. We managed to replace costly 32 bit floating point multiplier with 20 bit approximated fixed point multiplier and still the error and the visual degradations are negligible in terms of human vision perception. We estimated that the proposed solution shows 35.27% reduction in power consumption of the key multiplication operations in the NL-means filter.

A possible future work could be devoted to a deeper statistical evaluation of the proposed mutation operator, the use of evolved multipliers in other applications and a hardware implementation of the whole filter on a chip.

Bibliography

[1] Buades, A., Coll, B. and Morel, J.-M. Non-local means denoising. *Image Processing On Line*. 2011, vol. 1, p. 208–212.

[2] Češka, M., Matyáš, J., Mrazek, V., Sekanina, L., Vasicek, Z. et al. Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In: IEEE. *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, p. 416–423.

[3] Chippa, V. K., Mohapatra, D., Roy, K., Chakradhar, S. T. and Raghunathan, A. Scalable effort hardware design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. IEEE. 2014, vol. 22, no. 9, p. 2004–2016.

[4] Cilio, A. G. and Corporaal, H. Floating point to fixed point conversion of C code. In: Springer. *International Conference on Compiler Construction*. 1999, p. 229–243.

[5] Dorigo, M., Birattari, M. and Stutzle, T. Ant colony optimization. *IEEE computational intelligence magazine*. IEEE. 2006, vol. 1, no. 4, p. 28–39.

[6] Han, J. and Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In: IEEE. *2013 18th IEEE European Test Symposium (ETS)*. 2013, p. 1–6.

[7] Jiang, H., Han, J. and Lombardi, F. A comparative review and evaluation of approximate adders. In: *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. 2015, p. 343–348.

[8] Miller, J. F. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 1999, p. 1135–1142.

[9] Miller, J. F. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*. Springer. 2019, p. 1–40.

[10] Mittal, S. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*. ACM New York, NY, USA. 2016, vol. 48, no. 4, p. 1–33.

[11] Mrazek, V., Hrbacek, R., Vasicek, Z. and Sekanina, L. EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, p. 258–261.

[12] Mrazek, V., Sarwar, S. S., Sekanina, L., Vasicek, Z. and Roy, K. Design of power-efficient approximate multipliers for approximate artificial neural networks. In: *Proceedings of the 35th International Conference on Computer-Aided Design.* 2016, p. 1–7.

[13] Sekanina, L., Vasicek, Z. and Mrazek, V. Approximate Circuits in Low-Power Image and Video Processing: The Approximate Median Filter. *Radioengineering.* 2017, vol. 26, no. 3.

[14] Sekanina, L., Vasicek, Z. and Mrazek, V. Automated search-based functional approximation for digital circuits. In: *Approximate Circuits.* Springer, 2019, p. 175–203.

[15] Sze, V., Chen, Y.-H., Yang, T.-J. and Emer, J. S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE.* 2017, vol. 105, p. 2295–2329.

[16] Vasicek, Z. Formal methods for exact analysis of approximate circuits. *IEEE Access.* IEEE. 2019, vol. 7, p. 177309–177331.

Appendix A

Appendix

A.1 CD content

- Tech report (tech_report.pdf)
- file with source codes (source_files.zip)
- README - user manual for software content

A.2 Additional real life images denoised by approximated 20-bit multiplier

Figure A.1: Noised image with noise parameter $\sigma = 50$

Figure A.2: Denoised image with mul20s_p500 multiplier

Figure A.3: Noised image with noise parameter $\sigma = 50$

Figure A.4: Denoised image with mul20s_p500 multiplier

Figure A.5: Noised image with noise parameter $\sigma = 14$

Figure A.6: Denoised image with mul20s_p500 multiplier

Figure A.7: Noised image with noise parameter $\sigma = 50$

Figure A.8: Denoised image with mul20s_p500 multiplier

Sources for the images:

https://digital-photography-school.com/tips-for-photographing-football-soccer/

https://besthqwallpapers.com/animals/husky-family-pets-puppies-cute-animals-64517

https://www.wallpapertip.com/wpic/oxbxoi_romantic-background-images-hd/