UNIVERSITY OF TURIN
POLYTECHNIC

Electronic Engineering, Telecommunication and Physics

Master degree course in Electronic Engineering

Master Degree Thesis

Hardware Acceleration for AI

Robust pose estimation and neural networks

Supervisors:
Prof. Andrea Calimera

Candidates
Giuseppe Calderoni
matricola: 251679

Internship Tutor
Prof. Iris R. Bahar

Academic Year 2018-2019

A mia nonna Adriana
che ha sempre creduto in
me.

Summary

The following document is the summary of six months of work and experiments conduced at Brown University Engineering Research Center under the weekly revision of the professor Iris R. Bahar. Hardware acceleration offers great flexibility in terms of data reuse and algorithm implementation. This thesis proposes to import into the FPGA world a high-level algorithm used to improve the reliability of the convolutional neural network for robotics, in particular in the area of the object and pose recognition. The thesis topic is to transpose the high level algorithm proposed in the paper GRIP[1] in to a synthesizable circuit that can run on FPGA, especially on Xilinx® Virtex UltraScale FPGA ZCU102 board, thanks to the Vivado® suite. First of all, the original algorithm will be analysed to highlight the defects, in terms of parallel computing, and the possible improvements. Secondly, a new computing architecture will be proposed and developed to maximise the performance, pointing out the data reuse inside the algorithm.

Acknowledgements

I would like to offer my special thanks to Professor Andrea Calimera, my thesis advisor at the Polytechnic of Turin, for giving me the opportunity to connect to this project and to be always available for any reason.
I would like to thank Professor Iris R. Bahar, my thesis advisor at Brown University, for accepting me, for the material she provided and for her infinite friendliness and good sense.
A special thanks goes to my parent, who have always looked after me and given me the opportunity to have great experiences during my life. Moreover, they have always supported me in my decisions allowing me to grow day by day.
I am deeply grateful to my girlfriend Carla for the support she has offered me in these almost seven years of relationship. She is a key point of my life and without her I could not have discovered different realities and made difficult decisions.
I would like to thank the rest of my family which always believed in me and for their unconditional love.
I would like all the fantastic people that I known during my university path that actively contribute to let me be what I am now and who I will be.

Special thanks to :

Edoardo, Antongiulio and Marco to be the best roommates ever.

Beatrice, Valentina, Alessia and Daniele to the strong bond we have.

Andrea L., Andrea P. and Riccardo to be the perfect "guest" family.

Enrico, Francesco R. and Gregorio to the great memories we share.

Bruno, Nino and Martina, to be fantastic as they are.

Alessandra, Danilo, Diego e Nicola to the time, friendship, successes and trust we shared.

Andrea C., Francesco S. , Manuel, Marvin and Massimiliano, to be the best friends I could ever ask for.

Federico, Gabriele, Marco Ca., Marco Co. and Lorenzo, to share great time spent together and to mutual support during this hard university career.

I would like to thank all my american's friends Elahe, Yanqi Liu, Felipe, April, Chris,Adriana,Eren Jiwon, Jarod , Jasmine and Brennan for the real good time spent together and the friendly atmosphere. I am particularly grateful for the assistance given by Yanqi Liu (J.), especially for the time and the afford that she gave for this project, to Elahe for being so helpful even when we didn't know each other.

Giuseppe Calderoni

Contents

Listings

Water is fluid, soft, and yielding. But water will wear away rock, which is rigid and cannot yield. As a rule, whatever is fluid, soft, and yielding will overcome whatever is rigid and hard. This is another paradox: what is soft is strong.

-Lao Tzu

Thesis structure
This short summary has the aim to describe briefly the content of each part :

1. In this first part the original paper is presented and explained. The main goals of the thesis will be described , the organisation to reach them will be developed and from the analysis of a simple version of the algorithm the main bottlenecks will be pointed out.

2. Background research : artificial intelligence, random number generation, 3D graphics, tools and FPGA.

3. The last part shows the hardware implementation, with a description of all the sub-modules, and the main results will be presented.

Acronym

- **NN** : Neural Networks.
- **CNN** : Convolutional Neural Network.
- **SMC** : Sequential Monte Carlo Method.
- **3D** : Three-dimensional.
- **FPGA** : Field Programmable Gate Array.
- **HW**: Hardware.
- **IC**: Integrated Circuit.
- **ASIC**: Application specific integrated circuit.
- **NRE**: Non-recurring engineering cost, which refers to the one-time engineering cost, as research, design, develop and test a new product.
- **CDF** : Cumulative density functions

Part I

Introduction

Chapter 1

Introduction

1.1 The problem

Nowadays, thanks to machine learning, robots are even more integrated into our lives. The demand for an autonomous scene understanding, even in complex environments, is considerably increasing.

To get a reliable understanding of the scene and, consequently, a right manipulation of robots in unstructured environments, according to the state of the art for artificial intelligence, is still a challenging problem.

The main cause of error in these systems lies precisely in unconditional trust to neural networks and specifically due to their inability to recover from false positives. In a real scenario, complex and adversarial environments, such as poor lighting conditions, opacity, occlusion and noise, makes them more vulnerable to errors. There are neural networks, like PoseCNN, able to guess the position of the object and its pose, but they suffer from the same problems described above. This weakness, instead, is the main ability of the probabilistic inference. Combining both methods it is possible to strengthen the robot's perception of the surrounding environment and correct the errors produced by CNN.

1.1.1 The paper aim

The paper GRIP[1] proposes to correct the weak points of NN problem with a two-stage approach, combining and getting the best from both generative and discriminative inference.

The input , coming from a RGB-Depth camera with a resolution of 640x480, is directly feed to the first stage, a convolutional network. The latter provides, as output, the probability, for each object class, for a specific bounding box . The latter is transformed through a post-process algorithm to a bidimensional heatmap of likelihood. Moreover, to reduce false-negative cases the output of the network

is also not hard thresholded. After, the second stage, which is a SCM or better known as Particle Filter, has the task to find the 6DoF of the object under test, by comparing the object's point cloud to the depth one. Although the second stage has the capability to find the pose of the object without the help of first, the execution time and the computational cost increase dramatically. The process is iterative and its ending is defined by the reaching of a certain number of iterations or a specific level of confidence. A further characteristics, which reduces the false positive, of the second stage is that it does not restrict its searching space only on the result of the first one.

As it is possible to notice from what described before, the main task of the first stage is to restrict the research area by locating them in the image and generate information, probability, about which objects are inside the scene.

In the original paper, the code was written to be executed partially on a GPU thanks to the API Nvidia® CUDA. To find the final pose a realistic model, in terms of physical dimensions, of the object under test is used to be compared with the depth scene. The architecture SIMD allowed the executions of multiple particles, 625, to improve the processing speed.

1.1.2 Taxonomy

Across the thesis, some words can show a specific or slight different meaning respect to its common one. To provide a clear understanding of the process is important to explicitly describe the sense of special keywords :

- **Bounding Box**: is a rectangle, defined in pixels, which bounds a portion of the input image
- **Heuristic**: It is the corresponding volume for a bounding box. In the algorithm, it is used to reduce the possible space in which the object could be.
- **Heatmap**: Two-dimensional probability distribution organised as an image, in which each "pixel" represent a bounding box in the original image. In this project the number of layers used is three, as like the image scale used.The first stage produces a raw version of the heatmap, which will be refined before starting the second stage.
- **Pose** : In robotics, or broadly speaking in computer vision, the position in the space of a body can be described by three angles $(roll, pitch, yaw)$ and three space coordinates (x, y, z).
- **Sample or particle**: It is a specific pose of the current object under test.

- **Base frame** : It is the point of view of the robotic arm in the rest position. This reference system is different respect to the camera frame, our input point of view.
- **Point cloud**: A collection of point in the space, which describes an object or a scene.
- **Inlier**: It is a value that specifies how much a object's rendered pixel is close to the depth information, inside the scene.
- **Weight**: It is a the final value that characterises how similar is a sample respect to the depth image and it is used to classify the particles during the particle filter.
- **Absolute index**: it refers to the index of the heatmap memory.
- **Relative index**: it refers to the index of the sample memory.
- **Diffusion**: It means to generate new poses close to the most plausible by adding a noise in their parameters.

1.1.3 The main algorithm explained

Since the final purpose is to identify the presence of specific objects and their spatial position respect to the robot a double analysis stage is used. Figure 1.1 represents the particle filter operations in detail:

1. The a preliminary step consist to feed the input frame to a pyramid convolution neural network, which will analyse the image in several scales and provide, for each of them a raw heatmap.
2. The output from the first stage is transformed into a bi-dimensional probability distribution by applying a normalisation across the three heatmaps.
3. The bounding boxes with the depth image are used to generate the heuristics. After, the heatmap is sampled with an importance sampling, to pick "high" confidence bounding boxes. the latter are used to generate the samples in the right location thanks to the heuristics.
4. Now that the samples are defined, the second stage can starts by comparing the particles with the depth image and classified them according to the similarity.
5. According to the output weights, the particles are sorted and filtered out from the particle filter.

6. If the most likely sample triggers the output condition of the algorithm everything finish, if not a new iteration is executed. For the latter condition before restart there is the need to repopulate the sample array with diffused particles.

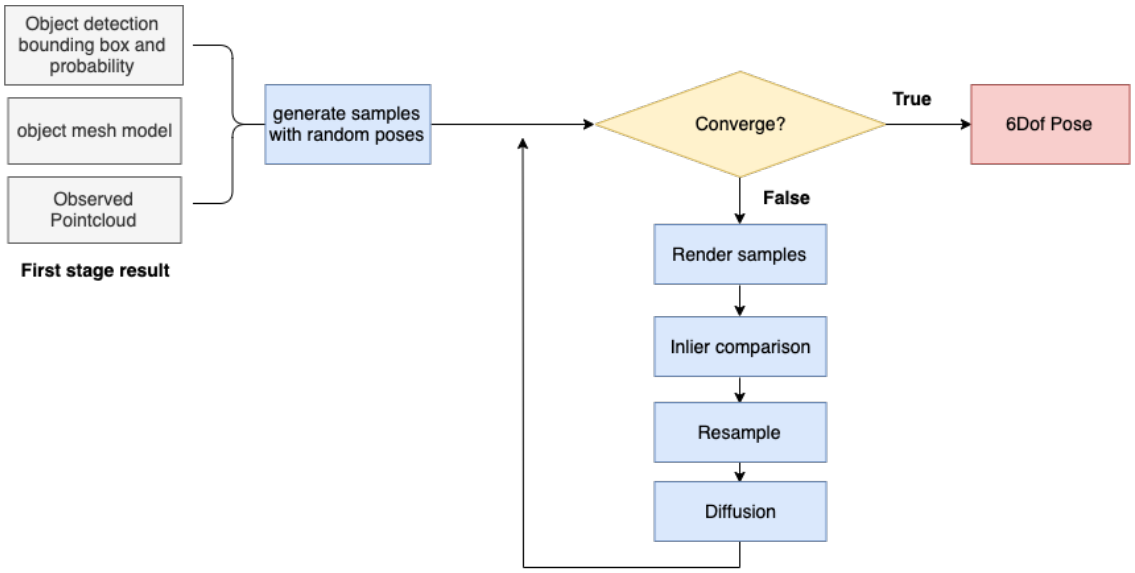


Figure 1.1: Flowchart of the second stage used in the GRIP paper.

1.1.4 This project goal

The programmable logic allows reaching the same performance or higher than GPUs thanks to the higher flexibility in terms of fast architecture exploration and easy testing. Unfortunately, the clock frequency is drastically different and to compensate this problem custom parallelism, concurrent modules, custom pipeline, special data type design and strong local data usage must be adopted to increase the performance keeping lower the power dissipation. The main goal will be to rethink the algorithm itself instead of a classical transposition to archive better results.

1.1.5 GANTT

A GANTT diagram is a specific type of diagram used to organise and represent the evolution of a project. Planning the time slots for each topic is essential in real projects, and this approach will be also applied to this paper.

- 15th - 16th May :Arrival in Boston and accommodation

- 17th - 20th May: Getting to know the city, the professor and the various colleagues.
- 21st - 31st May: Understand the status of the project, begin to analyze it and become familiar; with various tools.
- 1st - 30th June : Study the potential of the Vivado HLS compiler and perform experiments.
- 1st - 20th July: Complete the basic version of the program and analyse possible problems.
- 21st - 31st : July: Complete a first module in two different versions to understand the potential and the differences (one performed with finite state machines optimised and the other through Dataflow).
- 1st - 13rd August : break.
- 14th - 31st August : Bibliographical research.
- 1st - 30th September : Creation and optimisation of the individual modules .
- 1st - 24th October : Implementation and testing of the individual modules with an incremental approach.
- 25th - 27th October : Return to Italy.
- 28th October - 15th November : Thesis writing.
- 16th October - 20th November : Presentations and refinements.

1.2 Preliminary work

As said before, paragraph 1.1.4 , this thesis aims to rethink the algorithm initially developed to run on a PC/GPU platform. To attack the problem the algorithm will be deconstructed in small independent modules and reassembled in a complex architecture. To reach the goal a first hardware architecture was developed to understand and analyse which data, including their type, and operations truly matters and which partial results can be "recycled" among the same and across different mathematical operations, to reduce redundant calculations.

The base version was able to execute the same algorithm proposed by the GRIP paper. Even if both of them are realised in C++, the original one uses a very high level of abstraction and custom libraries, while the newer version does not. To find

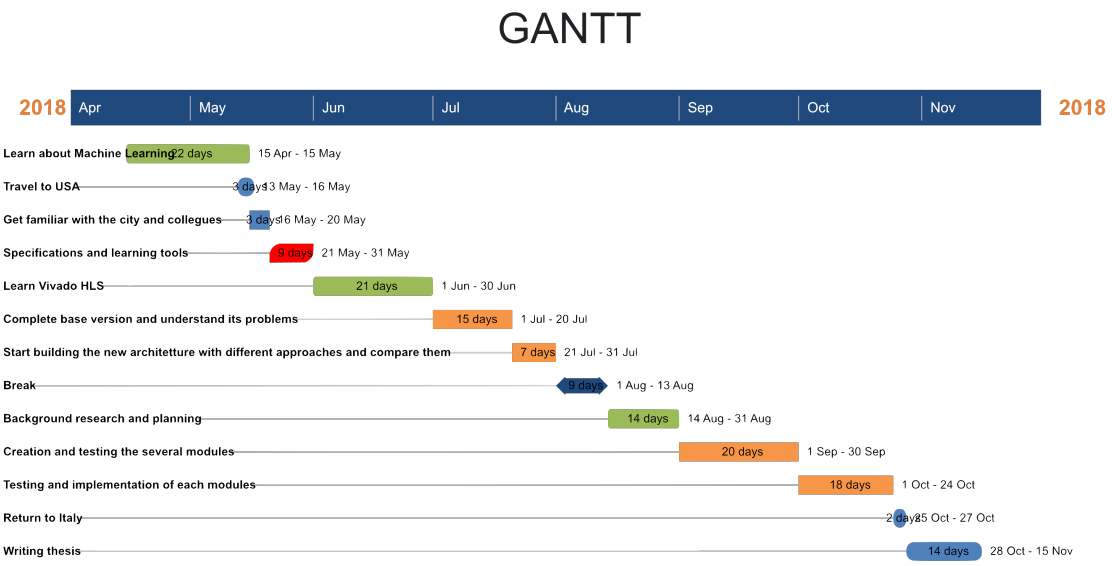


Figure 1.2: GANTT of this 6 months project

the shareable resources and bottlenecks the project was decomposed in a hierarchical view and analysed individually. The decomposition made the preliminary analysis easier and gave a broad view of which processes can happen concurrently. Furthermore, this modular approach, instead of a monolithic one, made easier the incremental testing, upgradability and individual optimisation.

The FPGA model used is Xilinx® Virtex UltraScale+ FPGA ZCU102 board, which include a external micro-controller supported by a OS, based in Linux, in which the data transmission is assisted and the interfacing with the robot is supported. The tool used to develop the system will be the followings :

1. Vivado HLS : a tool provided Xilinx® capable to convert C/C++ code to HDL, especially VHDL and Verilog.

2. Vivado : a tool provided by Xilinx® capable to interface with the FPGA, generate the bit-stream, route and program it.

3. Vivado SDK : a sub-tool used to program the micro-controller inside the FPGA, Zynq.

Difference respect to the GPU implementation

The preliminary version was developed with the following differences respect to the original :

- To execute the inlier function a slimmed-down rendering stage was developed to perform the object rasterization on the FPGA.
- The current algorithm do not more generates point clouds from depth image of the observed scene and the object model separately and then thresholds the Euclidean distance between the points in the two point clouds but analyse just the depth difference for all the useful pixels, if it is less than 5 mm that pixel is count as an inlier and increase its score by one.
- The generation of random numbers happen inside the FPGA

A more complete view for the second stage can be seen in the appendix B.

Computational complexity of the second stage

Fig.1.3 shows a summary of the computational complexity in terms of loops.

1.2.1 Where we can improve

From the base implementation of the basic version is possible to state that the algorithm itself is a bottleneck for FPGA implementation :

1. Only one object at time
2. Only one Sample at time
3. Only one Triangle at time
4. No Pipeline or effective structure for rasterization
5. No Pipeline for objects
6. No back-face culling in the rasterization
7. A lot of data transfers
8. No optimisation on packing the information
9. No optimisation for speed and area
10. Monolithic Design hard to reuse
11. Serial approach in most of the operations

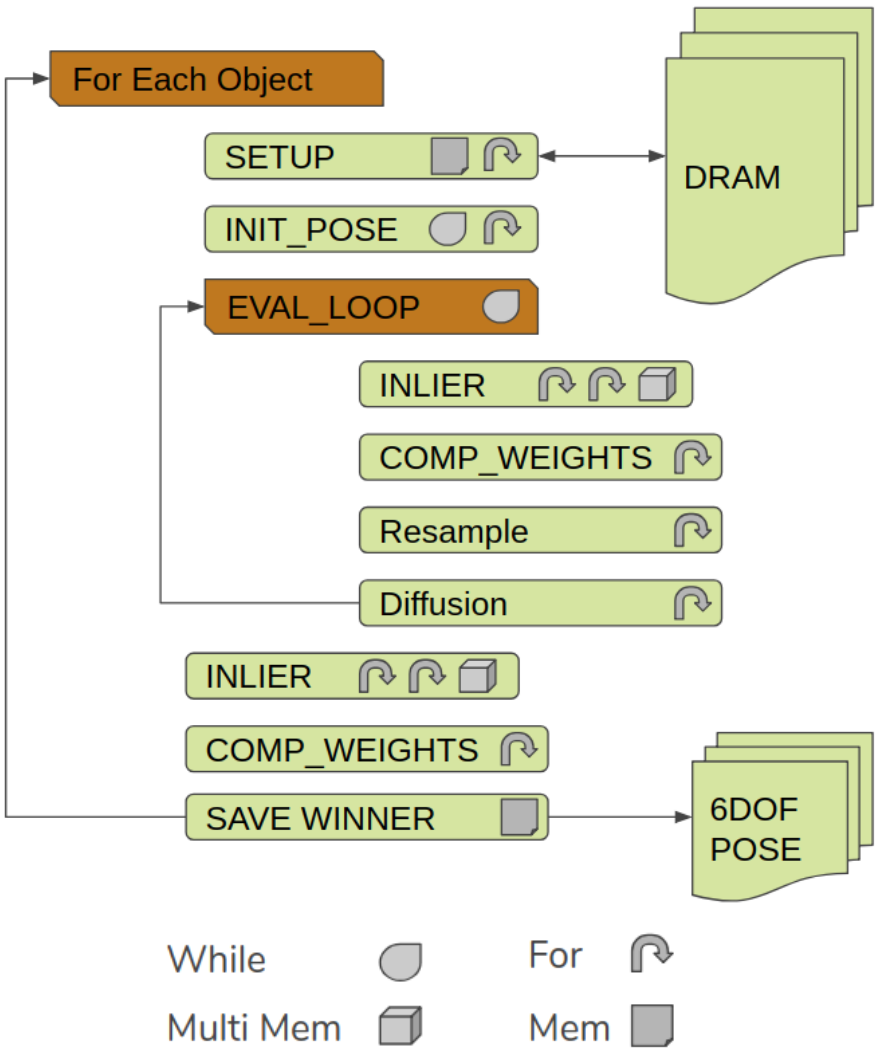


Figure 1.3: The algorithmic point of view with top computational complexity.

12. Current implementation uses C standard library to generate uniform random number

13. ROM with 1000 samples of normal is distribution is indexed by random number from the uniform random number to emulate random number generation from normal distribution

14. Floating point calculations

After performing the tests needed on the basic version, the major delays occur due to the sorting algorithm, the rendering, the comparison with the depth map

and finally the resampling stage. Even if the mathematical operation happens in parallel, this algorithm will not implement well in HW because the flows needs specific information before continue. The design is running on a 100MHz clock. The FPGA performs poorly in comparison in the rendering because it processes

cycles	Rendering	Inlier Function
GPU	3600000	31830000.0
FPGA	19000000	5451062.5
FPGA/GPU	5.278	0.171

Table 1.1: Performance GPU vs FPGA

each sample sequentially while the GPU renders all 625 samples in parallel. It does better calculating the Inlier function because it's only going over the bounding box and the calculation is reduced to a 1D analysis.

1.2.2 New hardware architecture

As a first step the problem was broken down in a hierarchical manner and consequently each sub-module was analysed individually, broken down again and re-elaborated with a specific architecture to archive minimum delay, less computations and lower memory usage, when possible. Since the algorithm itself is iterative, its not possible to parallel macro operations, the architecture development will focus on merging the operations and the creation of a set of systems that are self-sufficient and optimised. Moreover, the architecture is developed for architectural flexibility, which means that modules can be deployed and called independently. This approach allows a RTL level system construction and makes easier the testing inside the board.

A general view of the architecture developed is shown in Figure 1.4.

To achieve real-time processing the rendering process and the inlier functions are merged and a multi core rendering system. The transposition from floating point to custom fixed point reduced the hardware and power consumption with the decrease of the II. A faster, pipelined fixed-point matrix multiplication unit, reduction in memory on the FPGA used for particle-wise comparisons in the inlier function, and developing a new diffusion step to process samples quicker each iteration. About the sorting a custom, small memory usage, sorting network was developed and divided in individual modules to sort in the real-time as output for the inlier function is performed.

Broadly speaking, the whole project was developed in terms of data packets. Like in serial transmission, each data packet is redirected to the correspondent module and processed online while the processing element reads the next input. The figure

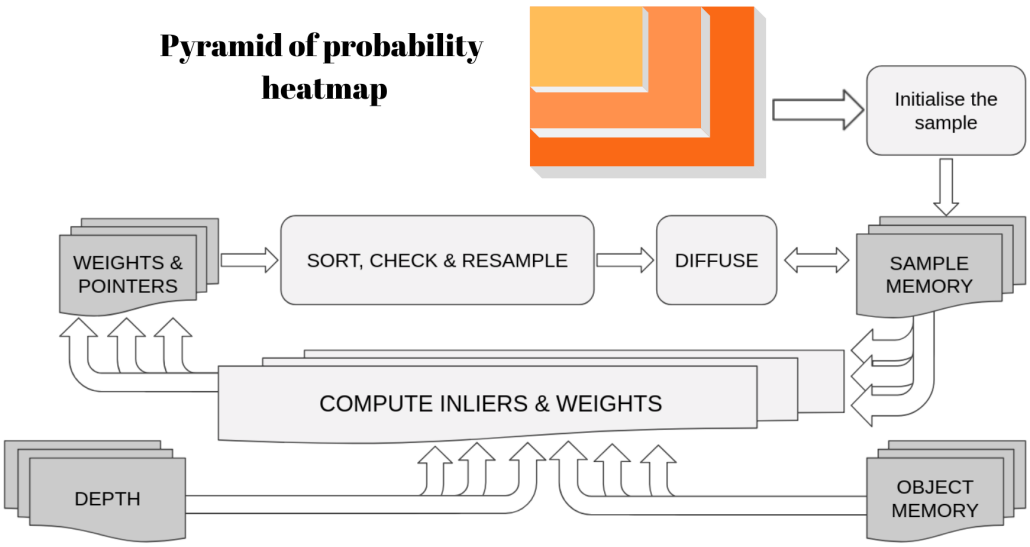


Figure 1.4: General architecture

7.1 shows the design flow used during the project design and testing. Image 1.6 and 1.5 are the result of the stage.

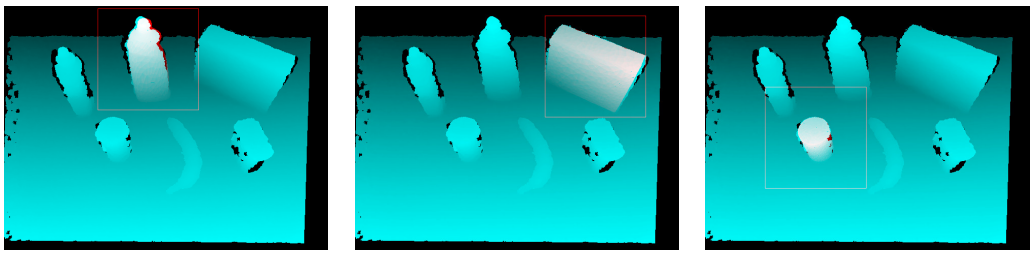


Figure 1.5: Scene and object model in the right pose, part 1

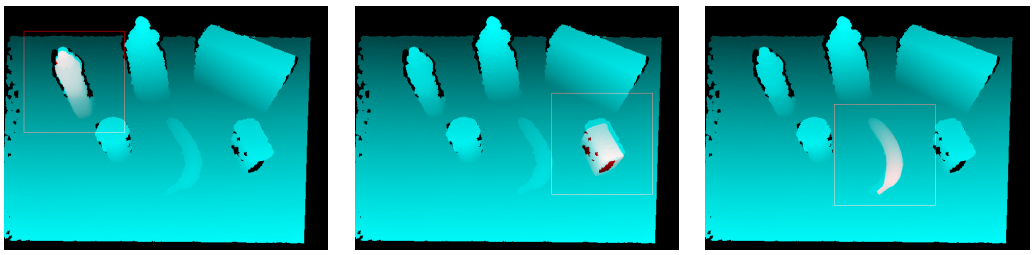


Figure 1.6: Scene and object model in the right pose, part 2

Part II

Thesis background

Chapter 2

Data driven algorithms

In an increasingly evolved world, full of complicated requests is necessary to have systems capable of interacting with non-standard and evolving environments. In recent years a gradual return to the use of neural networks, and more generally to artificial intelligence, was possible thanks to the availability of computational power, investments, by great giants of the SMART world, and by the improvement of the base algorithms used to built up this type of system.

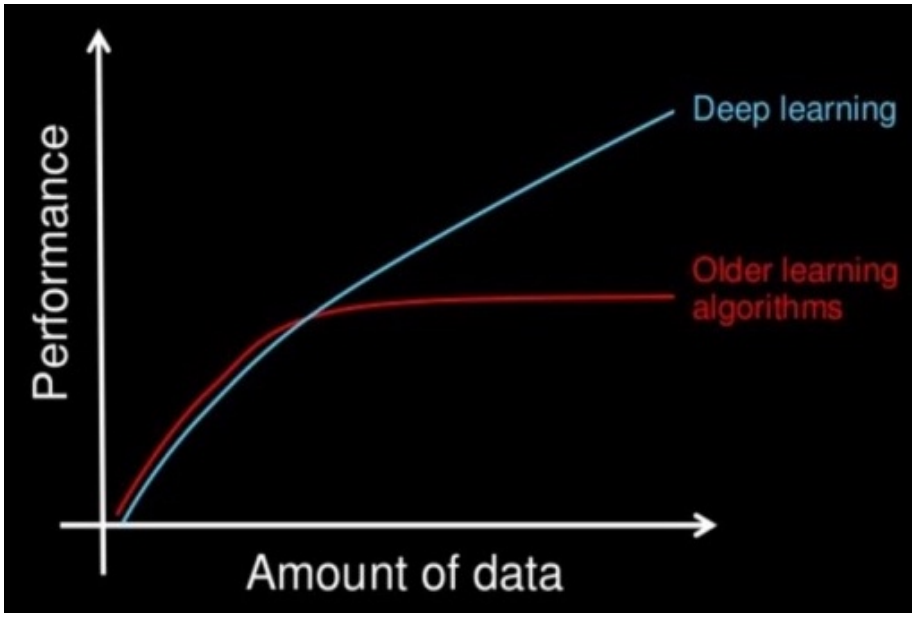


Figure 2.1: Performance of Deep Learning vs traditional algorithms according to the input size[5].

Being a data driven system means that the problem does not need a real formulation, rather it needs consistent data, which have to contain a good amount of

content and quality of information, to be processed. Artificial intelligence, in its more generic view, is able to solve strongly non-linear problems typical of the modern world, like " Big Data " or biological mechanisms . What will follow is a brief description of the concept that stands behind this argument.

2.1 AI, artificial intelligence

It is a generic term that indicates any program that can imitate the cognitive abilities, often associated with human beings. These skills are extremely important in the field of problem solving and learning. In general, therefore, an artificial intelligence builds in itself a set of generalised concepts that are used to solve a specific task according to the scenario. Although the possibilities and the results that can

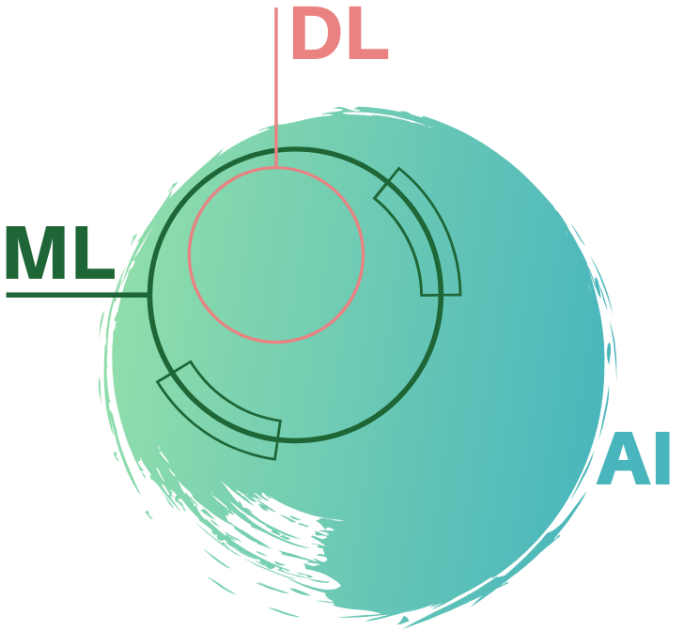


Figure 2.2: This figure shows the hierarchical order of the groups under the AI concept.

be archive with the use of AI surprise, they are not perfect yet. The type of artificial intelligence that the scientific community is mainly developing is also called "weak or narrow AI"[2]. This kind of AI is able to recognise patterns similar to the ones given during training period and classify the new ones in predefined categories. While, true artificial intelligence, the one specifically called "strong AI" is able to use "association" to process data, resulting to be more robust when subjected to adverse scenes, such as changes in light, opacity and occlusions. For the narrow AI this lack prohibits / restricts intensive use in complex environments.

2.1.1 ML, machine learning

Also known as Automatic Learning, it is a system capable of performing an operation without having been explicitly programmed, but it uses statistical methods, such as inference, to progressively improve the quality of execution. Each input is characterised by attributes, which can make sense to humans : " height ", "weight", "forecast conditions" and etc... and the system learn how to classify them. The process of feature extraction is often performed by humans, and is fundamental for the classification stage, 2.3. At the end of the development the aim is to built inside the machine an efficient mathematical model of the process under test, just by looking at the data. The general approach to create this type of machine usually consist to take the available data set and split it in two parts : training, supervised or not, and testing. The first is used to adjust the relative importance of each characteristic, while the second one is used to measure the performance and accuracy. Since the technique used to extract of characteristics and quality of a population is the statistical inference , the training set should contains non-redundant and informative data.

Figure 2.3: ML : features are extracted separately from the elaboration engine

What follows is a brief overview on how some of the ML models look for the "classification" :

- **Logical models** It's a type of module that use logical expressions to divide the input into segments and construct group models. Logical models can be listed in two major class:
  - Tree models : a tree graph is constructed to provide the relative probability according to a certain classification.
  - Concept learning : each attribute has a weight, without giving the sense of bad and good and the machine learn to recognise them by itself.
- **Geometrical models** Differently from logical expressions a geometrical model look at the features inside a geometrical dimensions, even if the attribute is not intrinsically geometric the importance is that it can be modelled to be like. With this kind of system the classification can be done with curves, 2D plane, surfaces, 3D space or distance measurement.

- **Probabilistic models and deep learning** These models use the probability of an event to classify new inputs. Broadly speaking exists two classes of probabilistic models :
  - Predictive : uses the conditional probability to predict an output according to a certain input.
  - Generative : estimate the joint distribution between the input and the output.

2.1.2 DL, deep learning

It is a subclass of machine learning, which means that everything mentioned above is also related to this subset. The peculiarity of this class is its ability to obtain the feature extraction of the objects during the training phase. These features, almost completely obscure to the programmer, will be used for classifying the input during the test phase. To emphasise the difference with machine learning, we can say that deep learning finds by itself the characteristic that serves to classify an element,Fig 2.4, while the machine learn needs an "external body" to indicate it,Fig 2.3. There is also a great difference in the number of data needed to grow and generalise a deep learning algorithm respect the case of machine learning. A further important information belongs to the taxonomy, because a neural network class is defined according to its depth, the number of hidden layers. If the latter is greater than one its name becomes DNN, deep neural network.

Figure 2.4: DL : the feature extraction happens intrinsically

The main tool of deep learning is the ANN, Fig 2.5 and eq 2.1, the artificial neural network, which has the ability to imitate / approximate any equation, $f(x_i) \equiv ANN(x_i)$.

$$out_k = \phi(\sum_{i=0}^{N} w_i \, in_i), \quad \phi \text{ Activation Function} \tag{2.1}$$

This ability can be interpreted as a derivation of the universal approximation theorem [6] or can be related to the study of probability, in particular Bayesian inference [7]. In the probabilistic case, the behaviour of the neural network can be interpret

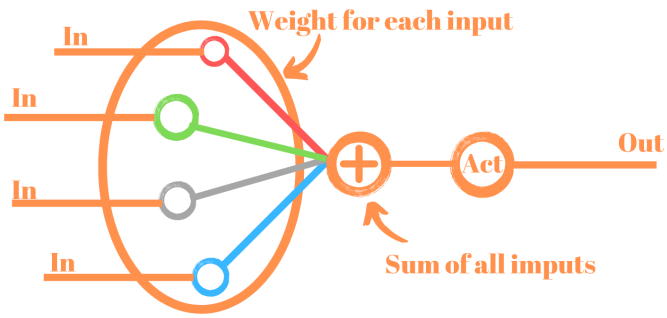


Figure 2.5: Structure of a neuron

as a hypothesis generator, with a certain level of trust, conditioned by the current level of knowledge. This value changes according to the data in possession and can be described precisely by the Bayes theorem, eq.2.2.

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)} \tag{2.2}$$

A probability distribution generated by the available data makes possible to formulate hypotheses on a given subgroup of phenomena. This value converges to the real value if the number of elements used to create the distribution contains a representative number of the entire population. Anyway in order to try to avoid the over-fitting of a characteristic exists several methods as the Dropout. During the training process in order to avoid the over-fitting of a batch, randomly across the epochs, some neurons are disabled. This method mimic the ability of our brain to have only a part of the overall neuron specialised to execute a certain function or recognise specific features.

The DL class has other very special property respect to a generic element of the AI world. Neural networks are capable of processing unstructured data such as sounds and images, respectively RNN (Recurrent Neural Network) and CNN (Convolutional Neural Network), extracting relevant information about them. Unfortunately, this capacity is offset by the power consumed due to the huge amount of computations.

2.1.3 CNN, Convolutional Neural Network

A particular interest in this project lies precisely in the convolutional of neural network, since it is capable of extracting information from extremely dense objects of information such as images, in fact it is extremely used in the area of computer vision. The model itself is developed based on the biological process that occurs in the visual cortex : edge detection and space localisation. Unlike a classic neural network, where each neuron is connected to all the others to form the next layer (fully connects layer), in CNN only sub-portions of neurons are connected to each other. The structure consists in four key elements : Convolutional layer, Non-linear calculation unit, pooling and Classification or/and segmentation.

- **Convolutional layer**: It has the task of dividing the image into small fragments and transforming the input them thanks to the presence of specific filters, better known as kernels. In itself, if the mathematical convolution operation consists in the integral of the product between two signals, in which one of them is translated within the integration domain. What really happens in this level is more like a cross correlation, which is extremely similar to convolution but conceptually different. In fact, the correlation represents a signal filtering operation while the cross-correlation represents a measure of similarity between the two signals, [8].

$$Convolution \ : \ (f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)\, g(t-\tau)\, d\tau \tag{2.3}$$

$$Cross - Correlation \ : \ (f * g)(t) = \int_{-\infty}^{+\infty} f^*(\tau)\, g(t-\tau)\, d\tau, \quad f^*(\tau) \text{ is the complex conjugate of } f(\tau) \tag{2.4}$$

  In the two-dimensional case, cross-correlation occurs through the use of a reference matrix called the relatively small $N * N$ kernel, like a Sobel one, that is scrolled on the image under analysis with a specific step, also called stride. In itself though this stage consists of a scalar product stage between matrices,fig 2.6.
  Each kernel, which scans the input image, produces a matrix called "feature map".
  The difference with the convolution in 2D is just that or the kernel or the output image is flipped, increasing the overall complexity.

- **Non-linear calculation unit**: Represents in itself the learning unit, the neuron, which calculate a "weighted sum" of its input and apply an activation function. This functions serves as a decision-making module. Selecting one

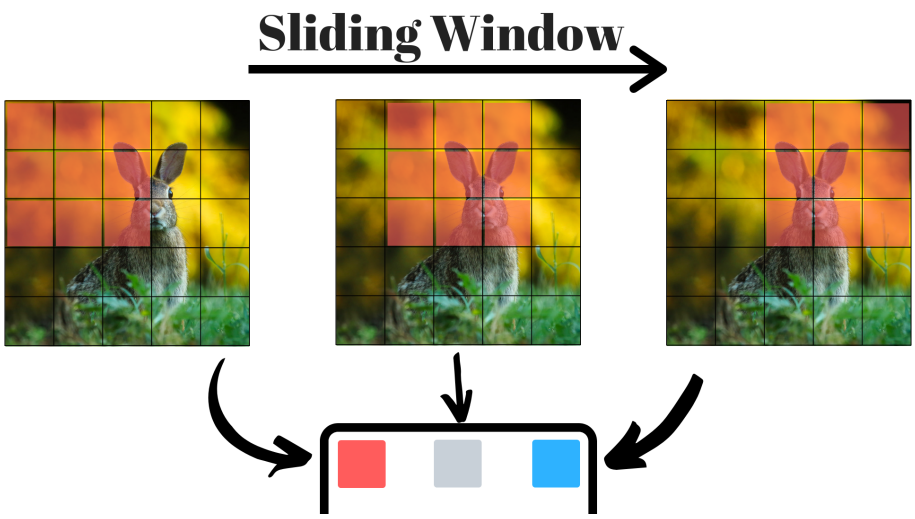


Figure 2.6: Creation of the feature map, cross-correlation operation.

type of activation function with respect to another alters the computational cost during the image analysis process. The most common are ReLU, Sigmoid, softmax and hyperbolic tangent.The neuron is capable of recognising complex patterns.

- **Sub-sampling unit** : It is also known as Pooling layer, makes the machine tolerant to geometric variations. This level reduces the size of the cross-correlation output matrix by combining multiple results from the same region. There are different methods of compression but the two simplest and most famous are the average and the maximum, where respectively the average is performed or the maximum is taken.

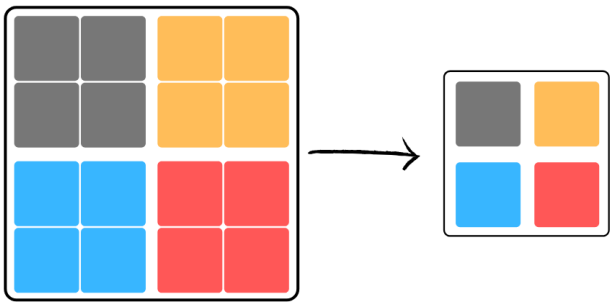


Figure 2.7: Feature matrix compression.

- **Classification or/and segmentation** The last stage of a CNN can be an FCL, a fully connected layer, which classifies the image within a series of predefined classes or a convolution layer,in which the kernel used is a 1x1 matrix. In the latter case, each pixel is classified by generating a mask that segments the various objects recognised in the input image. Due to the use of only convolutional layers the network takes the name "fully convolutional network".
  An example can be found in appendix A, it's a CNN with two layer of convo-

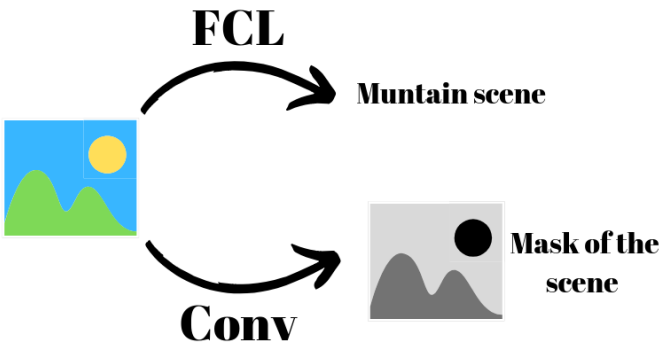


Figure 2.8: Difference in the output according to the last layer.

lution, two non linear calculation, two dense layer of neurons that converge in ten classes of classification. This method of visual recognition already in 2015 exceeded the human capabilities as can be seen from the image 2.9.

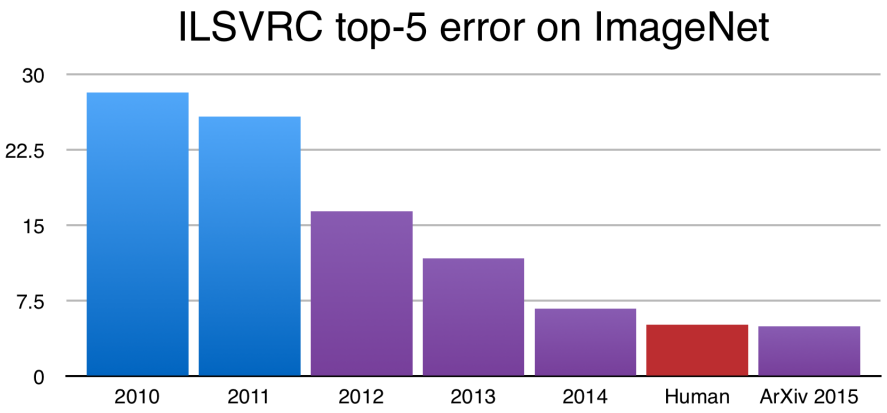


Figure 2.9: Neural network across the time versus human ability[9].

Pyramid Network: This particular type of CNN simply analyses the image in different resolutions. Changing the resolution involves a compression and therefore a loss / alteration of the information contained in the initial image. This means that some features are deleted, while others continue and persist becoming more pronounced. The neural network generates a series of bi-dimensional output , which contains the probability for each element in each bounding box for the different scales.

2.2 Particle filter

The Particle Filter is a method based on a predictive analysis and the estimation of internal stages in a dynamic system, when only partial information is available on it. This method can be applied if the process in question is a Markovian one, its future state depends only on the stage immediately present and not on the entire history of the system,[10]. In general particle filter can be used for a several task like localisation, SLAM or Fault Diagnosis. The goal of this stage in this thesis will be to determine the position in space (6DoF) through the following steps.

1. **Initialisation of the samples/particles** From the multiple probability density maps coming from the first stage the "importance sampling" method is applied,[11]. This sampling technique is a method for estimating the properties of a specific distribution having information from other distributions related to them. In this case the process consists in the creation of the normalised cumulative sum among all the distributions in possession. Once this new distribution is found, a uniform sampling is used to select the initial samples. Importance sampling is often used in Monte Carlo methods to reduce variance, since if a sample is more important with this method it will be selected more often in the sampling phase.

2. **weight calculation** In this stage the wanted object is rendered within the scene. In the paper version, this process takes place in 3D with the ICP algorithm, while in the HW version the analysis is reduced to the evaluation of a single dimension, the depth. In general, this stage calculates the geometric distance between the rendered object and the one present in the scene and if this value is less than a certain value we consider this measure satisfactory and assign a value. Each sample subjected to this stage will be assigned a "weight" related to the number of matches with the actual depth.

3. **Importance Resampling** For the selection of the sample to be introduced in the next cycle the same sampling process used in the initialisation stage is used with the only difference that in this case each sample is ordered according to its weight.

4. **Diffusion** To get more information from the next cycle we introduce some disturbances on the current samples, thus removing the redundancy in the set of samples. The types of noise introduced are of Gaussian type with zero mean and variable time variance and are applied to the position of the 3D model, partially alternating its 6DoF. Furthermore, the disturbance to the pose reduces

5. **Check the convergence** The process iterate till we found a value with an acceptable level of confidence, customisable, after that a new object is processed. In order to avoid endless run a maximum number of iterations is set.

Chapter 3

FPGA, field programmable gate array

The field programmable gate array is a specific type of IC able to be "programmed" according to the algorithm wanted. The FPGA , instead of using a fixed system architecture to perform predefined commands or operation as like the CPU, it consists in a grid of programmable logic connected via re-configurable connections, allowing the IC to be whatever needed: microprocessor, mathematical accelerator, graphic unit, digital filter or all of them concurrently.
Nowadays, the develop of a FPGA system takes always less time also thanks to the integration of third part cores.

3.1 Why choose it

The FPGA design compared to the ASIC for the production of a system shows different pros and drawbacks.
Referring to image 3.1 the FPGA in terms of entry cost, prototyping, time to market and NRE it drastically overcame the ASIC production. The main reason why it allows faster prototyping is that the synthesizable logic can be tested and implemented in the short term, allowing to work on the analogue and "software" parts concurrently
On the other hand, in terms of power consumption, analogue customizability, performance, clock speed, size and high volumes production the ASIC design provides the best.
As final consideration, with the programmable architecture is possible, differently form the ASIC design, updated the machine with new algorithms to solve bugs not visible in the test phase, to adapt to new standards or to reconvert it to execution different tasks.

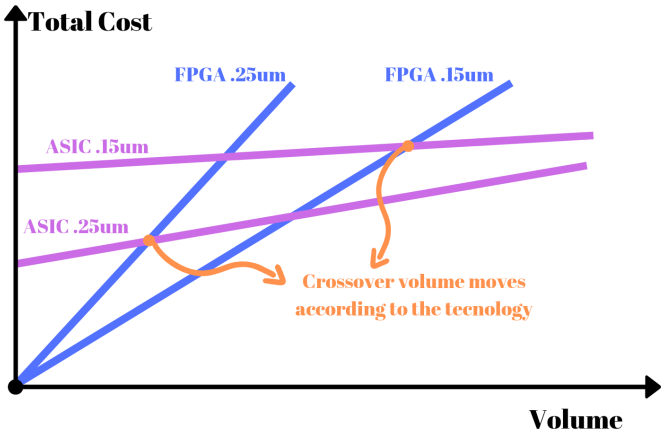


Figure 3.1: FPGA and ASIC cost vs volume, from [3] .

3.2 What is it made of

The fundamental building blocks of any FPGA architecture are LUTs, FF, I/O and programmable interconnections. In modern systems, there are also other components such as BRAM, High-Speed serial Transceiver, Off-chip Memory controller and DSP. These elements have been introduced to improve the interfacing and performance of the IC.

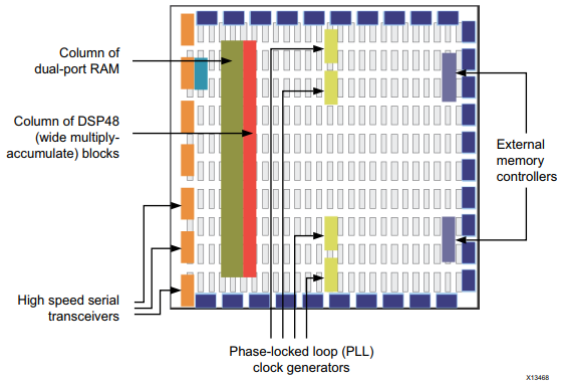


Figure 3.2: Modern architecture from Xilinx ®[15].

- **LUT (look up table) and DSP**

  It is the basic building block and therefore simpler block that can be found in architecture. The implementation of this block occurs through the use of multiplexers and re-configurable bits. In general, data x input the number of addressable cells is equal to 2^x and consequently, the number of representable logic functions is equal to $2^{(x^x)}$.
  FPGA, due to its degree of elasticity and parallelism, lends itself very well to running signal processing algorithms. These methods use elements called MAC intensively, that is multiply and accumulate, and logic functions/pattern detector. For this reason, in modern FPGAs, there are specific blocks capable of performing this operation called DSPs,[14].
  They are characterised by a pre-adder, a multiplier, a second logic lock that can function as an adder/logical operator and a pattern detector.

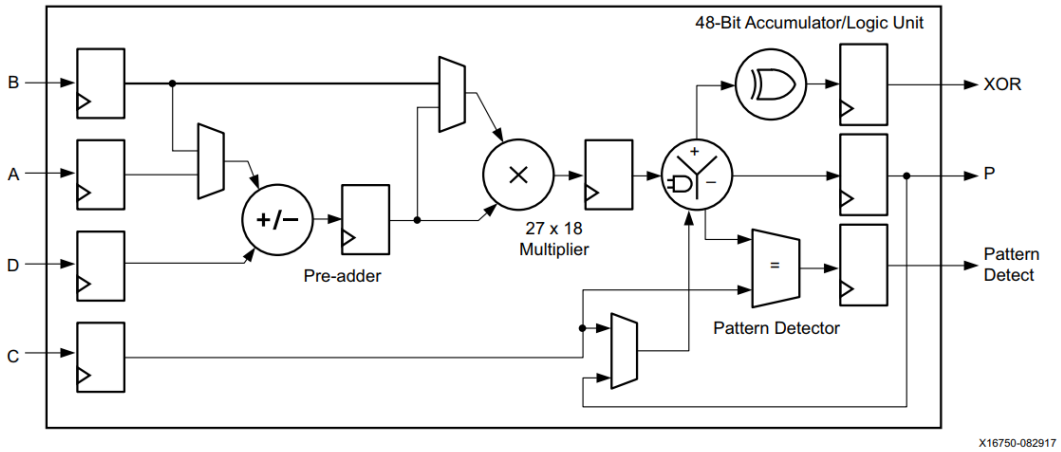


Figure 3.3: DSP48E2 functionality, from [14].

- **FlipFlop, slice and BRAM**

  The flip-flop is the simplest synchronous memory block that can be found inside an FPGA. If grouped together to contain information whose number of bits is greater than one, the FF vector, takes the name of Register.
  Flip-flops coupled with LUTs and multiplexers form a slice, which is the fundamental block of the FPGA. These modules provide the elasticity in the pipeline and storing processes, typical of these ICs.
  Furthermore, inside the FPGA, there are blocks of integrated memory, with random access, called Block RAM (BRAM). In the specificity, the type of BRAM present in the architecture of the VCU102 is a re-configurable 18K

block formed by two 9K blocks.
The 18K module, therefore, has 4 ports and can be configured to store information in the following sizes and access types: 16K x 1, 8K x2, 4K x 4, 2K x 9, 1K x 18 (M cells x bits) as True Dual port (RW / RW for each port) or 512 x 36 as Simple Dual-port (RO / WO),[13].

Figure 3.4 is a screenshot of a real floorplan, in which are present several slices, DSP and BRAM.

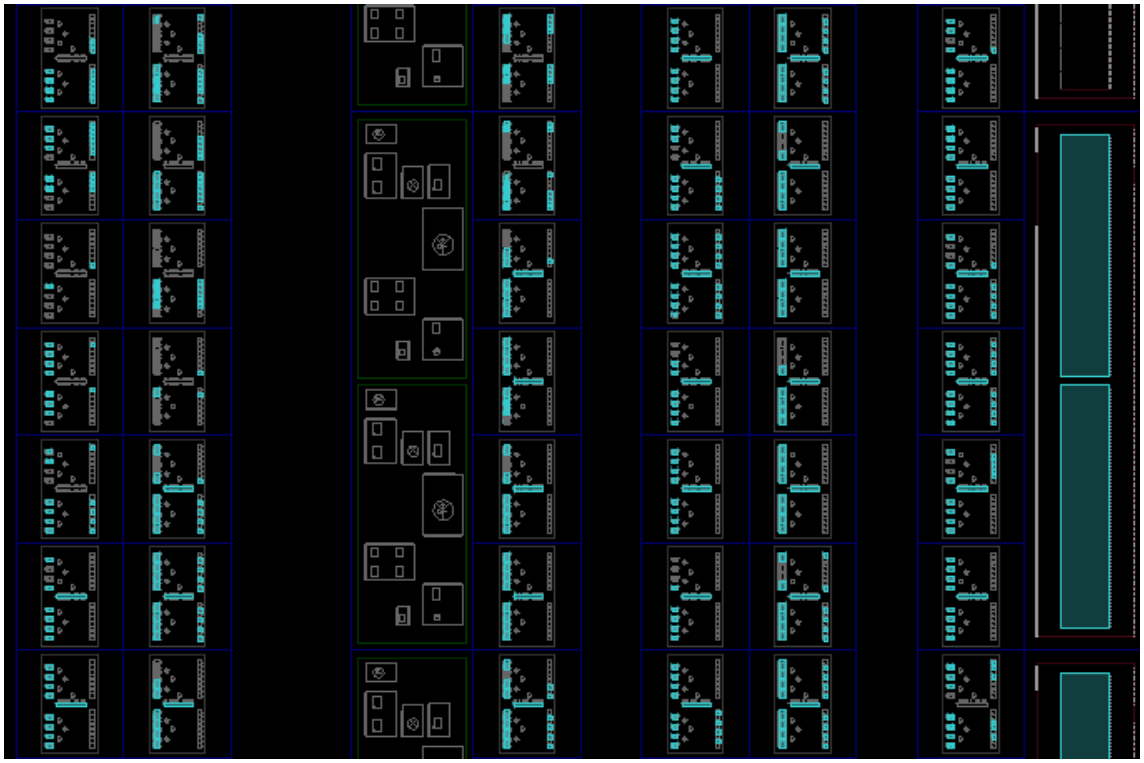


Figure 3.4: Floorplan FPGA ZCU 102.

- **Connections and Partial Reconfiguration**

  The re-programmable connections are the second strong point of the FPGA because through a series of programmable channels and exchange centres/nodes, implemented as pass transistors, it is possible to arbitrarily connect the various slices.
  Another capability of the modern FPGA is partial reconfiguration. If an external/internal processor is connected to the PL is possible to load/substitute part of the logic inside the module, these changes can be made without stop

the current ones.
This is possible just by load pre-compiled partial bit-stream files, exist certain design criteria to respect to use this property as like that the clock in that region must be static and not change between the various interchangeable modules.

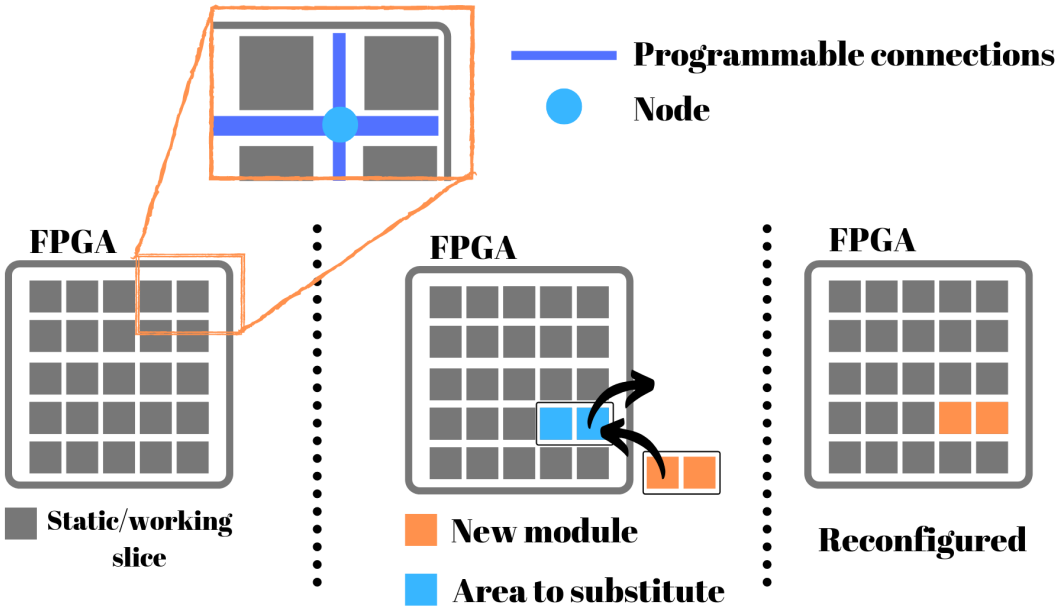


Figure 3.5: Reconfiguration process.

Chapter 4

Hardware Description Language and High Level Synthesis

In the past, when the digital design was very simple due to the low number of transistors inside the silicon portion, routing between the various components was done manually. Today, due to the needs of the market and the drastic reduction in the size of transistors, the digital design requires the development of devices that consume and cost less, but able to integrate and do more, making it impossible to perform routing manually. To contain this problem, hardware description languages , HDL, have been invented, such as VHDL and Verilog, provided to transform this process into a simple, clear, precise and formal language. This type of description allows the creation of a "netlist" that can be used by other programs, such as the engine of the place and the path.

HDL provides to design an RTL block, record the transfer level, in three ways:

- behavioural: describing the behaviour of the RTL block through the description of the process
- dataflow: describes the behaviour of the RTL block describing the logic circuit at the level of logic gates
- structural: describes the behaviour of the RTL block through the connection between blocks

This hardware description makes possible to create increasing complex and performing architectures, also, thanks to the possibility of reusing elementary blocks for multiple projects.

As mentioned above there are many engines capable of taking the netlist produced by the compiler and performing an auto routing or simply helping the designer during this phase. In general HDL languages can be used both for the description of

hardware for ASIC or FPGA. Precisely for this latter, thanks to its re-configurable architecture, HDL lends itself perfectly to prototyping making it an increasingly widespread system.

With the introduction of high-level synthesis, HLS, a subsequent step of abstraction is made, with respect to an HDL project. This step makes hardware design even simpler because it allows the designer to focus on the algorithm to be implemented and not only on the single register. However, HLS makes it possible to maintain extremely customizable hardware, through the use of a compiler instruction and consequently maintain a high efficiency from the circuit, shortening, even more, the design and testing. The supported languages are multiple C / C ++, System C, Matlab and Open-CL and most of their functions are synthesizable. This tool, therefore, brings with it a greater speed in architecture exploration, portability, simplicity in renewal and greater ease in debugging and post-analysis, thanks to the availability of C / RTL simulators integrated into the system. What will follows is a partial description of the functionalities used in the project, that are fully described in the [4]

4.1 Vivado HLS vs software development

Vivado HLS developed and tested by Xilinx® allows to transform C, C++, or SystemC projects in HDL code and simulate its RTL behaviour for most of the common application. The purpose of this tool is to be able to describe a HW circuit in a more flexible and maintainable way, which means that it has several limitation respect to a simple program written for processor, as dynamic memory , pointer limitation, recursion , system calls and virtual functions in C++ classes. As an example Vivado HLS during the synthesis needs to now exactly which is the size of every array because the concept of a dynamic memory has no meaning in hardware. On the other hand, some of the constraints introduced by the C++ language, as the fixed bits per type (int, char, etc ...), are useless and only degrade the possible performance of a logic circuit. For this reason, Vivado HLS provides two libraries to manage the arbitrary precision types of integers and fixed points so as to be able to maintain the desired calculation resolution and manage the required hardware, such as the number of DSP used or low level bit manipulation. What follows is a list of optimised libraries provided by Xilinx®:

- Stream: describes the stream class that will be analysed later
- Math: contains synthesizable mathematical functions and this library is used during C and C / RTL simulation to obtain accurate results.

- xfopencv/video: it is a set of kernels optimised for FPGA based on OpenCV and the Line-Buffer class for raster scanner algorithms typical of video processing.
- IP: contains the C equivalents of the IP cores provided by Xilinx® such as FFT, SSRFFT, FIR, DDS and shift register class
- Linear Algebra: like the IP library it contains a series of functions optimised for the linear tree.
- DSP: contains mathematical functions optimised for intensive DSP use
- SQL: contains optimised functions the stream class to execute SHA224 / 256 and bitonic sorting.

4.2 Vivado HLS Workflow and Synthesis

The way of designing with Vivado HLS generally follows the following steps:

1. Pre-design: behavioural/algorithmic description of the module to implement, set the timing constrains and consequently define the I / O and a possible high-level structure.
2. Set the three fundamental constraints of the project: clock frequency, FPGA model and frequency uncertainty.
3. Write, compile, execute and verify the behaviour of the program written in C++ or high supported languages.
4. Apply directives to improve current the design, making it similar to the one thought at the beginning.
5. Synthesizer the design and check the timing.
6. Check the conformity of the synthesised system through a C/RTL test bench, if possible and in case of error correct the design.
7. Create the IP package and import it in Vivado.

To produce a correct design it is very important to understand how the compiler creates the synthesizable code, because, although many processes are intrinsically controlled and optimised, it is up to the designer to provide the right directives/inputs to the compiler. By default, what is produced by the tool is a data-path controlled by an FSM, which reflects the behaviour of the code written in C++.

The number of states varies according to the frequency of clk and as a consequence also the generated data-path. By default, the type of coding for the finite state machine is "one hot", but it can also be set to graycode, binary and auto-optimisation, which starts from one hot and performs optimisation of the states. Vivado HLS has the ability to switch from FSM to micro-architecture if the number of states exceeds a certain number and as every micro-architecture, it also will perform the optimisation of the lifetime of the variables to recycle hardware. These synthesizable codes follow very specific standards:

- **VHDL** uses the IEEE 1076-2000 standard
- **Verilog** uses the IEEE 1364-2001 standard

Another feature of the tool consists in the creation of a set of drivers to configure/interface with the generated module, also called IP block, from a microcontroller.

4.2.1 Performance metrics

Almost all the synthesised logic is described by a series of fundamental parameters:

- Latency: how many clk strokes it takes to show the last value leaves the function
- Initiation interval: indicates after how many clk strokes the synthesised machine is available to process another set of input

In addition to this RTL top level view information we have information on the behaviour of internal logic such as:

- **Complete loop latency**: that is, how many clk it takes to place a loop let go of its last value
- **Cycle latency**: that is, how many clock cycles it takes for a loop to be complete
- **Cycle initiation interval**: i.e. how many clock cycles it takes for a loop to accept a new data

Sometimes these parameters are replaced by "**?**", if special directives, as **TRIPCOUNT** or **LATENCY**, are not added. For example in a loop with variable length will not have defined the latency because it depends on a variable.

4.3 # pragma HLS

After having described the behaviour of the circuit in c, it is important to proceed to the optimisation of the circuit in terms of access to memory, dependency declaration, throughput and parallelism. Each processing element generated in the top view can be optimised independently of the others. To reach the wanted design Xilinx® proposes a series of #pragma, or compilation directives, which serve precisely for this purpose.

4.3.1 PIPELINE

It is a digital design technique similar to that used in modern processors to maximise the throughput. Instead of waiting for II, the system will add in the correct position a block called buffer, which is a register, across or among the code operations, to process more inputs concurrently. This pragma can be applied both to at function and at loops level, with the only difference that in the first case the pipeline is executed forever while in the second case it lasts until the loop loads the last input. Note for the function pipelined, if there are no more valid data the chain stops until a new valid data arrives. To avoid unwanted pipeline stall we can specify the II to be different from one.

- **II** : is a parameter of the pipeline pragma which allows us to control how many clock cycles must pass before load a new value. This value is the most important in a chain, because it guarantee that the chain is always full. If the external world does not provide the correct timing for the function, bubbles were introduced. These bubbles are just useless calculations and strongly degrade the performance of a pipeline chain.

- **Rewind**: is an option of the pragma applicable if the function is called several times, as in a loop, to superimpose different iterations so as not to have to wait for the pipeline chain to empty.

- **Flushing**: This function, applied on a function, allows to become empty and therefore to produce the desired results even if there are no valid input data. This function is used extensively in the creation of state machines in vivado.

```
1 void burst_memory(volatile int * in){
2 #pragma HLS INTERFACE m_axi depth=depth_array port=in
3 #pragma HLS INTERFACE s_axilite port=return
4
5  int BRAM[depth_array];
6 //creates a burst access to memory
7 //if the location is sequential
```

```
8  for(i=0; i < depth_array; i++){
9   #pragma HLS PIPELINE II=1
10   BRAM[i] = in[i] + 100;
11  }
12 }
13 void adder(a_Data_T  & a, b_Data_T & b , c_Data_T & c){
14  #pragma HLS PIPELINE II=1
15  c = a + b;
16 }
```

Listing 4.1: Pipeline example

4.3.2 Dependencies

This pragma is strongly correlated to the pipeline, because as in a processor, in the moment a pipeline exist, will exist a dependence in the data inside the chain. To have a working chain is very important tell to the compiler if exists or not the dependence between several iteration or among the same. Typical dependencies occurs if read and write operations are executed.

- **Read After Write** , RAW, a value calculated in the previous instruction is used in the current.
- **Write After Read** , WAR, a value is overwritten in the current instruction, but its original value was used in the previous operation.
- **Write After Write**, WAW, among different instructions the the value is written and read, so that the chronological order matter.

The compiler can understand most of the time if exists and which type of dependence occurs, but sometimes it can be too strict, creating false dependencies. In case of loops if the dependencies belongs to different cycles, the selector "**inter**" must be used, while if belongs to the same loop iteration "**intra**" must be adopted. If the pragma dependence is used you must also specify if it the correlation is "**true**" or "**false**".

4.3.3 Loop merging, flattening and latency

Loops are very used in logic, because allows to execute tasks in a periodic and bounded way. By default to access and exit from a loop one clock cycle is wasted, so two pragma exist to take care about that.
The pragma **merging** will merge different loops to decrease the number of clock cycles. To merge two or more loops they must have bounded values, there must no

FIFO access and multiple execution of the same loop must generate the same result. If these requirements are fulfilled the compiler will generate a unique FSM with a unique loop, which bounds depends on the longest loop.
As we said to pass from one loop to another a clock cycle is wasted, and this is also valid for nested loops. Assume three loops nested named L3, L2 and L1 with the same length N and assume that the inner loop operation takes only one clock cycle to execute its body. We have for each loop one clock cycle to enter and one to exit, that means that to enter in L1 and exit from L1 we lose 2N cycles. The same value is also valid to enter and exit from L2, over all $4N^2+2$ are used to exit and enter in the loops while only N^3 were effectively useful.If the compiler is not able to merge the loops code must be restructured or manually merged. The pragma **flattening** will automatically collapse the three loops in just one. To respect timing and avoiding to create bottleneck the latency of a function or a loop must be controlled. To take this role is the pragma **latency**, which forces the compiler to archive the latency wanted. The pragma can set to archive at **max** or at **min** a certain number of clock cycles and can be applied both to functions and to loops with the only difference that if it is applied in the function all the region stays under this rule while if it's inside a loop body only the latter is subjected to this rule.

4.3.4 Unroll, allocation and function instantiate

Controlling the hardware instantiate from the synthesizer is very important because allows the designer to reach better performances. The pragma "**unroll**" is used to create multiple instances of a body region, act like a "GENERATE" in VHDL. This directive, used most commonly inside the loops, allows to reach lower latency at the cost of more area, which means that must be used carefully. The parameter that can be configured is the factor, which states how many instances must be created, and according to its value we can have a loop that it is partially unrolled, if factor is different from the number of iteration, or fully unrolled, in which factor is equal to the number of iteration. As said before it can also act like "GENERATE" in VHDL, which means that we can use it to force the compiler to produce the hardware wanted, like connections or modules. In the same way, that we want have the ability to instantiate more hardware, we would like to control the number of operators or functions instantiate, for this purpose the directive "**allocation**" must be used. This pragma allows to control the number of functions, core or mathematical operator by setting its limit number, it is very useful when a "if-else" statement is present or we want limit partially the hardware when the **unroll** pragma is present. The "if-else" is mapped in two independent modules which shares the output, example if a DSP is used in both the compiler will synthesizer two of them, even if only one will work. Another case in which the number of functions wants to be controlled is when the "**inline**" pragma is applied. This directive dissolve the external barrier created from

the function, allowing the compiler to optimise the logic. A further optimisation can be applied to the functions with the pragma "**function instantiate**", which allows to simplify the logic inside a function according to a input variable and the pragma "**resource**", which allows the developer to choose the core and the latency/pipeline stages for that specific variable.

4.3.5 Array partition, reshape and map

To maximise the performances, i.e to not be stuck due to the memory I/O, or optimise the memory allocation, without modify the code a series of pragma exists : **Array partition**: Normally the vectors are stored within BRAM, which have only two I/O ports. This could be a bottle neck if the algorithm needs multiple accesses for continuous readings and writes. therefore Vivado HLS gives the possibility to re-partition our array in several ways. The vector from the point of view of C++ is not divided but gains in terms of access during the synthesis, obviously this pragma increases the number of resources used. There are three basic partition parameters:

- **Block**: the vector is divided into K elements of equal size, in this version the cells that were originally consecutive remain consecutive
- **Cyclic**: the vector is divided into K elements of equal size, in this version the cells that were originally consecutive are distributed in the new memories
- **Complete**: the array is divided into its elements. In the case of multidimensional arrays we can specify which dimension is subject to this partitioning through the parameter dim, example array [N] [Q] [T] can be divided into four dimensions:
  - 0 → divided into registers,
  - 1 → array [N] [Q] [T] is partitioned into N vectors of size [Q] [T],
  - 2 → array [N] [Q] [T] is partitioned into Q vectors of size [N] [T]
  - 3 → array [N] [Q] [T] is partitioned into T vectors of size [N] [Q].

Array map : This pragma has the function to merge two or more array in only one. The need advantage of this pragma is that the designer do not need to merge manually the arrays, manipulate them in a bit level. The best use of this pragma happens when an array doesn't not fill an entire BRAM, but only a part, instead of wasting that space the developer can concatenate two array of the right dimensions to archive only one that fits better in the memory. Due the variable size of the inputs, two methods of concatenation exist :

- **horizontal** : concatenate, sequentially, the arrays in only one which depth is the sum of the arrays and the width is the maximum between the ones.

- **vertical** : concatenate, in parallel , the arrays in only one which depth is the maximum between the ones and the width is the sum of of the bits concatenated.

Exist another parameter called **offset**, which introduces N empty cells , according to the parameter value, to distanced the two consecutive arrays.

Array reshape : combine the vertical mode of the array map pragma with the array partition to concatenate elements. This pragma if well used can increase the parallel data access and decrease the number of (B)RAM needed.

4.3.6 Dataflow and stream class

As most of digital design, made of several functions, in order to produce the final value sub-operations are needed. These sub-operations create temporary results that will be used in the following stages to provide the the final value. In general, the temporary results are available in different time slots and the next module, that will use the temporary value, don't need to wait the previous module to end before start to work. The **dataflow** pragma allows to create concurrent processes able to start working in the same moment that a valid input is available. This kind of task level pipeline allows to archive higher throughput and reduce the latency.

> Quote [4] pg 148 from line one :
> "Dataflow optimisation potentially improves performance over a statically pipelined solution. It replaces the strict, centrally-controlled pipeline stall philosophy with more flexible and distributed handshaking architecture using FIFOs and/or ping-pong buffers."

- **Dataflow canonical region** Xilinx® recommends to add the directive inside a region, that will be called "canonical region" . The latter can be the body of a loop or can be the a simple body of a function. The connections between the functions will be called channels and they are modelled as FIFOs or Ping-Pong buffers. The **dataflow** pragma respect the boundaries of the function, unless the directive "**inline**" is present in the function inside dataflow region, because it will inline the function to the top one. Since the dataflow respect the function limit, each module can be designed and optimise to archive the time constrains wanted. Notice : If the dataflow is used in a loop and the functions inside the dataflow region are pipelined, the pragma "rewind" will allows to overlap several external iterations, reducing even more the final latency. To use this directive a series of constrains must be respected :

1. In the code, the connection channels must be a local, intended in the canonical region, non-static scalar / array / pointer or static **stream class** variable.
2. In the canonical region no feedback is allowed, only data forward algorithms.
3. Arrays can be read and write in only one process, one producer one consumer rule with the writer before the reader.
4. Functions must be void, the information must pass through channels.
5. No loop-carried dependencies among different processes via variables.

Disable start propagation
It is an additional statement that can be added to allow an unbounded slack between producer and consumer, having process inside the canonical region running forever only driven by data. Xilinx® at page 156 specify that this statement is at user's risk.

- **Stream class**
  It is an optimised class provided by Xilinx® able to simulate a stream of data, like a serial transmission. In the high level code it can be interpreted as a FIFO of infinite depth, having all of its constrains : sequential read and only one read. This class is the most suitable to model data transfer timing, each stream<type> variable can be read or write only in one process as happen for the dataflow arrays. Exist two types of reading and writing :

  – **Blocking** : stall the process if the input is empty or the output is full. Optimum to be used if no data wants to be lost and the data coherency is important.
  – **Non-blocking** : this allows the execution of the process even if a read or a write is attempted and failed. In this mode is very important to archive the II programmed and correctly size the depth of the channel, to avoid the lost of information. This mode differently from the **Blocking** is not possible to validate through C/RTL simulation.

  This kind of class can be used in non-Dataflow region or in Dataflow region, the difference is that if it's used in the first one to ensure a correct working the depth must be specified and correctly sized, otherwise the c simulation will stuck in a deadlock situation. The vantage in the non-dataflow region is to avoid the address request during sequential access, typical of a BRAM. The depth in the dataflow region should be sized according to the II and latency of the modules, otherwise deadlock. In general, the dataflow pragma allows to

have lower depth because the channel is filled and depleted continuously from the producer and consumer. **Note*** is mandatory that all the stream channels inside the same project have different names.

4.3.7 Custom FSM

In several case a full control of the FSM is necessary, i.e. to drive custom data-path created in VHDL or Verilog or to create/manage custom or standardised protocols. Large protocol system which manage data-packets maybe need more than one simple FSM to archive the final computation. Also in this case, the use of an HLS tool-chain allows faster development and architecture exploration, thanks to the higher abstraction control flow. The final FSM will contain extra logic respect to the base one because of the handshake between modules and starts and ending signals, which are almost transparent to the developer. The directive dataflow can be applied to execute all the FSM concurrently and the stream class can be used to simulate also in this case the connection channel. The structure of a FSM must be able to trigger a change in the input each clock cycle, which means that the II of each module inside the dataflow region must be one and the depth of each channel must be sized according to the latency of the module.

```
void FSM_Complete (hls::stream<struct> &PackT_In ,hls::stream<type>
    &PackT_Out) {
 #pragma HLS dataflow
// Interface is an axi4 stream
#pragma HLS INTERFACE port=PackT_In axis
#pragma HLS INTERFACE port=PackT_Out axis
// Declare the channels and their depth
static hls::stream<struct_One2Two > One2Two_stream
#pragma HLS STREAM variable = One2Two_stream depth = 4;
static  hls::stream<struct_Two2Thr >  Two2Thr_stream
#pragma HLS STREAM variable = Two2Thr_stream depth = 10;
// Declare the connections
FSM_1(PackT_In, One2Two_stream);
FSM_2(One2Two_stream, Two2Thr_stream);
FSM_3(Two2Thr_stream, PackT_Out);
}
```

Listing 4.2: FSM TOP example

```
 void FSM_1(hls::stream<struct> &PackT_In,  hls::stream<
   struct_One2Two > & One2Two_stream) {
#pragma HLS INLINE off // must respect the boundaries
#pragma HLS pipeline II=1 enable_flush // archive II=1 to be able
   to change state each clock and read input
// enable_flush is needed otherwise the FSM will stall if no new
   input is present
```

```
//...
}
```

Listing 4.3: FSM example

The FSM can be constructed with a switch construct, most common and clear way, or with if else statement, the tool will synthesise the same circuit with the only difference in the readability of the C++ code. Vivado HLS intrinsically assume that the condition **IF-ELSE** are uncorrelated and so it will create two independent different functions, and this can be a waste of resource especially if the function has II = 1. The pragma allocation can help to solve the problem, but must take into account of this problem when a custom FSM with embedded data-path is designed. Another trick that can be used to better explicate the current state is through the keyword "**enum**", which allows to associate a name to an integer.

```
static enum FSM_1_States{M_IDLE = 0, M_LOAD} FSM_Curr_Status;
```

Listing 4.4: enum example

With this keyword the readability of the code is improved. Since the stream class is used in this case the use of non blocking is, advisable.

```
void FSM(stream<typeIN> IN , stream<typeOUT> OUT) {
// all the declarations needed
...
// The body of the function
typeIN IN_fill;
// the read can be performed also inside the switch condition
if (!IN.empty()) {
 inData.read(IN_fill);
switch(currStatus) {
  case IDLE:
    memory = IN_fill;
    counter = 0;
    // SELECT THE NEXT STATUS
    currStatus = LOAD;
    break;
    case LOAD :
       memory += IN_fill;
       // SELECT THE NEXT STATUS
       if (counter < 9){
            currStatus = LOAD;
            counter++
       } else {
            currStatus = IDLE;
       }
    break;
    default:
    currStatus = IDLE;
    break;
```

```
29      }
30      }
```

Listing 4.5: enum example

4.4 I/O handshakes

Each system can be described as a black box in which only the behaviour is known and how it transmits/receives information and commands. This description is necessary when creating a complex system and wanting to introduce it into the processing chain. Vivado HLS creates three types of ports:

- Clock and reset: ap_clk and ap_rst respectively. There is also the possibility of a Chip enable, which can be configured in the config_interface entry
- Block-level interface protocol: ap_start, ap_done, ap_ready and ap_idle.
- Port Level interface protocols: ap_return plus all inputs and outputs

4.4.1 Block-level interface

This type of interface describes when the IP can begin to execute the operation when it ends and when it is ready or inactive. This type of handshake can also be specified if the IP block we are designing is of the empty type. if the return command is not specified in the C code, it will not be synthesised. The three types of Block Handshake available are :

- ap_ctrl_hs: it's a classical hand-shake between modules. The start is asserted, the module is triggered by the signal asserting the idle signal to indicate that it is operating. The ready line become assert when all the inputs have been read, and the done signal become assert when the module finish leading to the de-asserting of the idle signal.
- ap_ctrl_chain : it's the same as ap_ctrl_hs with the exception of an extra handshake wire ap_continue, which states to the previous module if the next module is ready for a new data or not, preventing the previous module to generate extra data . This kind of block level I/O is useless in dataflow region because is the availability of the data will lead to the synchronisation or stall.
- ap_ctrl_none: will not create any of the synchronisation in the ports. This mode is very useful if the project contains a series of blocks inside a region where the dataflow pragma is declared, not allowed if the dataflow is inside a for loop or contains sequential or pipelined FSM , because it reduces its rigid synchronisation.

4.4.2 Basic port-level interface

As the modules also the signals, whether they are variable or other type, the exchange of information about their status is needed to act consequently. Vivado HLS provides a complete set of protocols for the I/O, in order to give to the designer a better control of how the information pass through the modules. At page 82 of [4] Xilinx® provides a table to explicit which handshake is available according to the type and which is the default I/O handshake. The port level interface can be reduced to the following classes:

- **No I/O protocol** : in this class no I/O protocol is present.
    1. ap_none : is the default for the scalar, do not require extra hardware because it has no signal that states if the value is read or written. This lack of overhead can be useful if the information is provided at the correct time and held for the correct time
    2. ap_stable : is to be used only for parameter that change in the reset status. From the hardware I/O point of view is the same of the above with the implicit condition that its value is guaranteed stabled for all the function time
- **Wire handshake**
    1. ap_hs : is a intra-module basic handshake based on the signal acknowledge and valid. the valid is asserted when the value is ready to be read or written and the acknowledge signal will be asserted when it is read leading to the de-assertion of the valid signal.
    2. ap_vld : Act as ap_hs without the ack signal.
    3. ap_ack : Act as ap_hs without the vld signal.
    4. ap_ovld : Act as ap_hs without the ack signal but can be used only for output.
- **Memory interface**
    1. ap_memory : this protocol is used to implement array in memories in block memories (BRAM) with data, address, write-enable and chip-enable signals. The interface is implemented with discrete ports
    2. bram : same as ap_memory with the only difference that the signals are grouped inside a single port
    3. ap_fifo : act as like as bram memory interface with less hardware overhead, instead of address, write enable and chip-enable the ports implemented are full and empty, useful during the read an write access. The

access is only sequential, no address generation is needed, and the assigned port can be used only to read or write, not both. This is the default implementation for stream connections in the top level interface.

4. ap_bus : is a specific interface that do not follows a standard protocol. This can be connected to a bus bridge if a standardised communication is needed. This is the default interface for pointers and pass by reference and can allow normal r/w process or burst.

4.4.3 AXI4 port-level interface

Advanced Micro-controller Bus Architecture, AMBA, designed by ARM includes the Advanced Extensible Interface, AXI, since 2003. This open-standard protocol is mostly used in system on chip architecture to connect between themselves the different functional modules. Xilinx® encourages the use of this protocol to connect in the proper way the IP cores.

1. stream : It's a point to point connection, in which the data transfer happens in a sequential way and can be written from one function and read from only one function, if there are no present crossbars. The information is always extended to the next byte, registered to avoid combinational feedback and can be with or without side channels.

   - without side channels : it means that what you are transmitting is just an array.
   - with side channels : it means that what you are transmitting is a structure. Being extended bytes the structure can be compressed in just one channel with the pragma **data_packet**, which allows or a bit field padding or a structure padding. This directive also reduce the number of handshake signals. Often this interface is coupled with a DMA to directly write and read from memory without interrupt the micro-controller.

   Moreover, the handshake of this kind of port I/O can be also registered completely, partially or not.

2. light : It allows the IP core to be controlled by an external Interface, often used with CPUs or micro controllers. For the latter a special set of C drivers are provided by the Vivado HLS to control the module directly from the C code of the micro-controller.

3. master : This interface allows to write directly into DDR memory without pass through a DMA or Micro-controller. It can allows single or burst data transfer up to 256 data transfer cycles with just one address request. This

kind of interface needs also the base address to read and write from memory and the maximum size of the depth of the vector, the default base address is 0x00000000. The keyword **offset** can be applied to the pragma to add an offset.

(a) off : no offset is applied to the base address

(b) direct : 32 bit port is added in order to let the module apply the offset

(c) slave : thanks to the axi4 light interface is possible to set a registered offset information.

Notice that this kind of interface allows the module to access to any of the address location, and if the access is done with pointer in the Vivado HLS code the keyword volatile must be added. The **burst** mode is reauthorized by several parameters

(a) latency : which allows the design to initialise a bus request a certain number clock cycles before the r/w expected.

(b) max read burst length : specify the max number of data read during the transfer

(c) max write burst length : specify the max number of data write during the transfer

(d) num read outstanding : max of number read request to the bus without a response. If the number is exceeded the design will stall.

(e) num write outstanding : the same as for reading but for writing request

Chapter 5

Random number generators

In the Monte Carlo methods, security, cryptography and several other functions and discipline use intensively random numbers. By definition a random number generator can belongs to two different families :

- **Real random number** : In this class no prediction about the next number can be done according to the current value, neither if the structure of the generator is known.Real random numbers are preferable from pseudo ones but they are not always available because a specific hardware is needed to generate them. This hardware is capable to measure and sample a physical phenomena, as thermal noise, and correct external biases due to the environment or the circuit itself. One way to generate real random number is to lead a transistor in its metastability point, by violating its setup and holding time during the switch, and let the thermal noise decide the direction of the switching action.

- **Pseudo random number** : This class called pseudo because it is a deterministic series of number that looks random, and once known the current value and the structure of the generator is possible to determine the next and the followings. This set of algorithms can generate directly pseudo random number, as LFSR, with a precise distribution or can use several distribution and mathematical operations to generate the wanted distribution.

Thanks to its use a series of tests are available to check the type and the conformity of random number series. The easiest is the histogram, which shows information about the frequency of the numbers, but also more complex methods can be used, as [17].

5.1 Linear feedback shift register

It is a shift register, with N registers, in which the next input is evaluated through a combinational feedback. The input of the feedback are the output of some registers, called also taps. The most common way to create the feedback network is to use a xor or xnor port to collect all the taps and the output will feed the new input. The maximum length of the sequence follows the maximum number representable $2^N - 1$, the zero or one is removed, respectively in the XOR and XNOR implementation, because it would stop the feedback chain. From the mathematical point of view a LFSR is a polynomial, binary, in which the taps represent the equation. The number of register can be counted starting from one, the input, till N, the last element. In order to get a polynomial of N grade the feedback must include always the N^{th} register.

$$p(x) = 1 + b_1\, x + b_2\, x^2 + b_3\, x^3 + ... + b_N\, x^N\,,\ b_i$$

can be 0 or 1 in order to mark the presence or not of the i^{th} grade

In general to produce the maximum number of combination the taps must respect

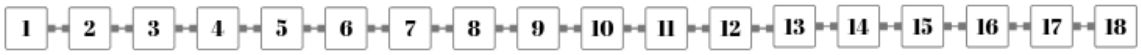


Figure 5.1: Register's numeration

the following rules :

1. the number of tap is even.
2. The greatest common divider between them must be one, they must be mutually prime.

According to the grade of the polynomial there must be more than one combination that gives the maximum number of combination. A table with the taps can be found in [18] . This LFSR is used in several modules in the second stage, like in the resampling stage. An example with eighteen bits is provided.

$$p(x) = 1 + x^{11} + x^{18}$$

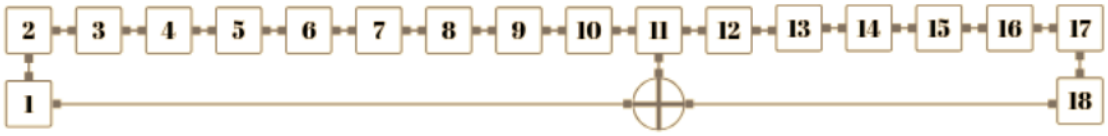


Figure 5.2: LFSR for 18bit

Inside the following example code the indexing in the array is flipped respect to the standard cell numeration and both the load of the input seed and the output is parallel. To not produce all zeros, a preliminary checks for the input seed is applied, if it does not respect the non zero condition a default seed is used. Fig 5.3 represent a simple cell.

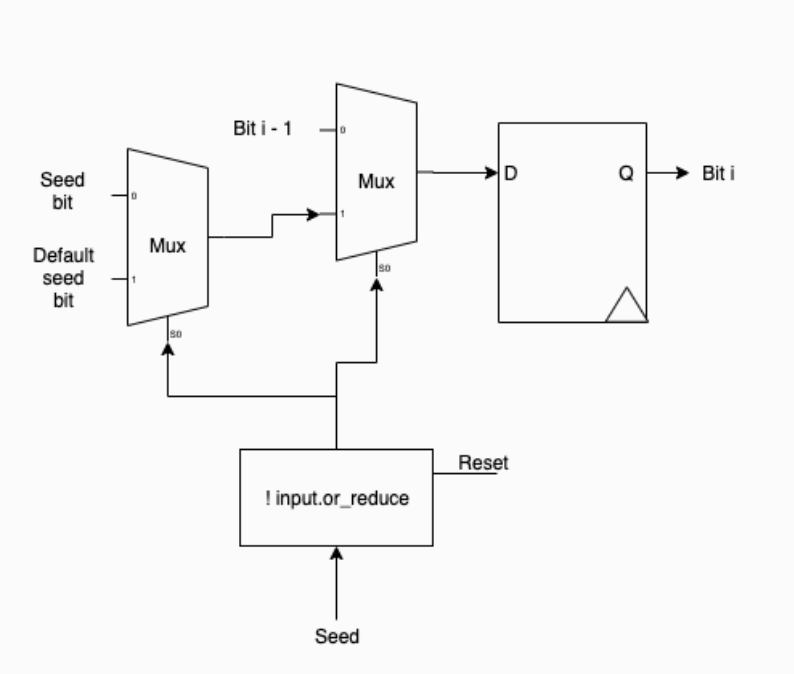


Figure 5.3: LFSR cell

Moreover, the generation of the random number is described as a process of a bounded length. This fixed number allows the system to consume less power, since the LFSR does not continuously produce uniform random numbers .

```
void LFSR18_init(u18_Data_T & seed, stream<u18_Data_T> &
    RAND_stream) {
  u18_Data_T lfsr; // u stands for unsigned and 18 stands the bit
    width
  if (!seed.or_reduce()){
    lfsr = seed_default;
  }else{
    lfsr = seed;
  }
  for (u10_Data_T i = 0; i < NUM_PARTICLES; i++) {
    u1_Data_T b_18 = lfsr.get_bit(18 - 18);
    u1_Data_T b_11 = lfsr.get_bit(18 - 11);
    u1_Data_T new_bit = b_18 ^ b_11;
    lfsr = lfsr >> 1;
    lfsr.set_bit(17, new_bit);

    RAND_stream.write(lfsr);
  }
}
```

Listing 5.1: LFSR example

5.2 Box Muller Transformation

Developed, by George Edward Pelham Box and Mervin Edgar Muller in 1958 [19], to produce a couple of independent number distributed in a normal way, Gaussian with zero mean and one as variance.

The method has been chosen to its flexibility in terms of mean and variance modulation and for the fact that a lot of operations can be reused. Furthermore, the Box Muller generate two independent Gaussian numbers as need by the diffusion module.

$$\begin{aligned} Z_1 &= \sqrt{-2\log(U_1)}\sin(2\pi U_2)\ ,\ U_i \in Uniform(0,1] \\ Z_2 &= \sqrt{-2\log(U_1)}\cos 2\pi U_2\ ,\ U_i \in Uniform(0,1] \end{aligned} \tag{5.1}$$

From equation 5.1 the generic $Z_i \in Normal(0,1)$.

$$Z_i' = \mu + \sigma Z_i \tag{5.2}$$

With the transformation applied in equation 5.2 $Z_i' \in Normal(\mu, \sigma^2)$ A more

suitable form to evaluate the cosine, instead of using another CORDIC module, from the sine can be using the trigonometric functions

$$\cos 2\pi U_2 = \sqrt{1 - \sin(2\pi U_2) * \sin(2\pi U_2)} \tag{5.3}$$

In this way we can recycle the square root module and have a fixed delay, differently from the CORDIC,[16], using less hardware. Moreover, if the algorithm needs only the positive half of the Gaussian distribution is possible to merge the square roots and reduce overall the computational cost.

```
void Box_Muller(stream<Usquare_Data_T> & U_square, stream<
    angle_Data_T> & angle_stream,
     stream<Gauss_Data_T> & Gauss_number,stream<Gauss_Data_T> &
    Gauss_number_2) {
  for (u11_Data_T i = 0; i < Gauss_number_needed; i++) {
    angle_Data_T angle_fill = angle_stream.read();
    ap_fixed<23, 5> U_square_fill = U_square.read();
    ap_fixed<23, 5> U01_log;
    if (U_square_fill != 0) {
      U01_log = hls::log(U_square_fill);
    } else {
      U01_log = 0;
    }
    ap_ufixed<18, 4> U01_fix = -U01_log;
    trigono_Data_T U01_sin = hls::sin(angle_fill);
    ap_ufixed<18, 4> U01_log_radical = hls::sqrt(U01_fix);
    trigono_Data_T U01_cos = hls::sqrt(1-U01_sin*U01_sin) ;
    ap_fixed<21, 4> Gauss_fill_h = U01_log_radical * U01_sin;
    ap_fixed<21, 4> Gauss_fill_h_2 = U01_log_radical * U01_cos;
    Gauss_Data_T Gauss_fill = Gauss_fill_h;
    Gauss_Data_T Gauss_fill_2 = Gauss_fill_h;
    Gauss_number.write(Gauss_fill);
    Gauss_number_2.write(Gauss_fill_2);
  }
}
```

Listing 5.2: Box Muller example

Chapter 6

3D Graphic Pipeline

The main part of this algorithms makes its fundamentals in 3D graphics. The algorithm in the inlier function needs to compare each object model, in a specific pose, with a depth portion. To archive this task the 3D model, of the object under test, have to be rendered and projected in the correct spatial location In modern GPU the operation to transform object models from 3D to 2D space is implemented in a "3D graphic pipeline". The latter is a sequence of mathematical operations to represent correctly a 3D object in a 2D space, like an image. The GPU graphic pipeline is made mainly from the following steps:

- Point Transformation : each object point is subject to a transformation, translation, rotation or/and scaling.
- Triangle setup : three point that are designed to make a face of the object are called.
- Clipping and back-face culling : cut the part of the triangles out of the field of view and the same direction of the camera view.
- Rasterizer : the triangle is filled to give the idea of a solid object.
- Depth buffer creation: a second frame in which the object is define by its distance from the camera.
- Fragment Processor : unify different fragments to create the final image.

Some steps like the texture mapping are not essential in this project, due to this they will be not analysed. The pipeline optimised for hardware implementation will be proposed later, chapter 7.

The object model To understand how the graphic pipeline works an introduction on how the objects can be stored and represented can be meaningful. In

general, an object, as every other shape in the space, can be seen just as a collections of surfaces connected, which can be represented in several ways :

- The perfect description of that surface can be obtained by the explicit equation but is not the best in terms of mathematical and computational complexity.
- Another way to represent a model is use basic polygons to construct an approximated version of the object, and as every approximation, if the number of polygons to describe the object grows the difference respect to the original model reduces to zero.

In computer graphics the most common way is the polygon representation. Across all the possible polygons the most dynamic is the triangle, because it is the simplest 2D shape. Each triangle or face is described by three vertices or points in the space, the bounded area represent one of the face of the model. According to the resolution needed the model can be adapted, by reducing the number of triangle needed, to reduce the memory consumption and make faster the chain. To have a realistic match with the depth information the object model and the object in the scene must be the same.

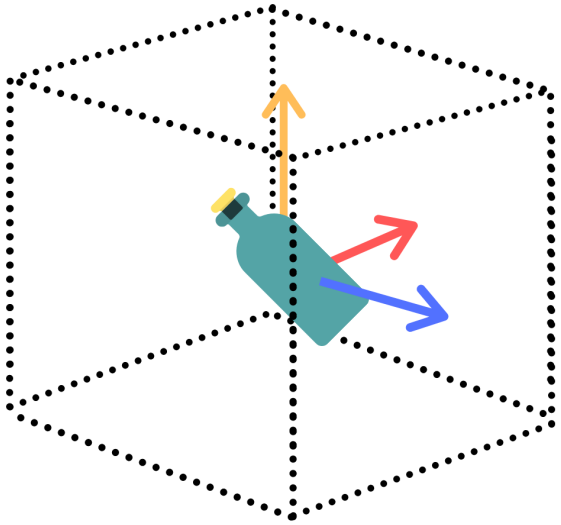


Figure 6.1: Object model space

The reference systems In 3D graphics usually are define several reference systems : the world space , object local space,The Camera space, NDC and the screen space. The first is the absolute reference for all the object in the scene,

camera included, while the fourth defines the portion of the space, visible related to the camera field of view.

- **The Local/Object space** : each object, described in a separate file is designed in its reference system. Its origin is in (0,0,0) and all of your object points refer to it, figure 6.1 .
- **The World space** : all the object models that compose the scene after the transformation,belongs to the same origin. In this place the scene is constructed.
- **The Camera space** : the world coordinate are transformed in to the view space, the point of view of the camera
- **The Clip space** : after the applying of the projection matrix the frustum is remapped in the Normalized Device Coordinates (NDC), this is called clipping space because it's very convenient clip in this space , special algorithms exists . Once we decide the frustum size and shape each point out will have a value higher than one, so easy to clip.

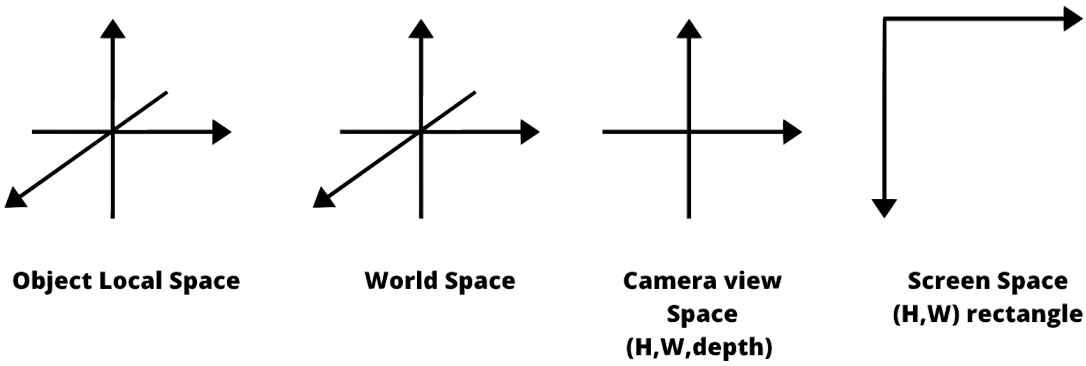


Figure 6.2: Some of the reference systems in the 3D graphic pipeline

6.1 Rigid body transformation

Since the object model is described with the real dimension, to match the camera view, a rigid body transformation is needed. A rigid transformation is a transformation that preserve the original shape of the object. In general the object model, to be rigid transformed, needs its origin in the (0,0,0) and exists three possibles modifications that can be applied to a body to change its space location and orientation.

- Translation, Fig 6.4.
- Rotation, Fig 6.6
- Scaling

To position the body in the wanted position the operation must be performed in a fixed order : the rotation, the translation and scaling. If the object is non zero centred it is not guarantee that the final pose is in the wanted location. To solve the problem the object model must be multiplied by the inverse matrix of the original position, to recenter it, rotate and translate again adding the extra translation needed.

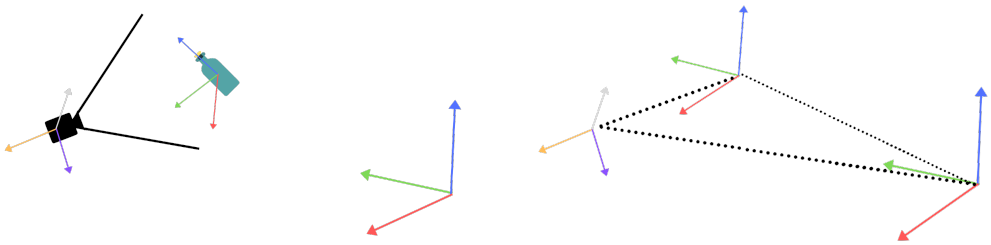


Figure 6.3: The object in the space with the two main reference systems

6.1.1 Three dimensional translation

To perform a translation in the space an offset vector must be added to the generic point in the space :

$$\vec{\mathbf{V}}' = \vec{\mathbf{V}} + \vec{\mathbf{Tr}} \tag{6.1}$$

More in general, since an object is described by multiple coordinates it correspond to add to each of them this offset vector. This will move the barycentric of the object from the origin to the offset point and in a homogeneous way all the other

points will be moved and without deforming the object. Since a matrix form will be useful in the future is better to define it now :

$$\vec{\mathbf{V}}' = \underline{\mathbf{Tr}}\vec{\mathbf{V}} = \begin{bmatrix} 1 & 0 & 0 & x_o \\ 0 & 1 & 0 & y_o \\ 0 & 0 & 1 & z_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_o \\ y + y_o \\ z + z_o \\ 1 \end{bmatrix} \tag{6.2}$$

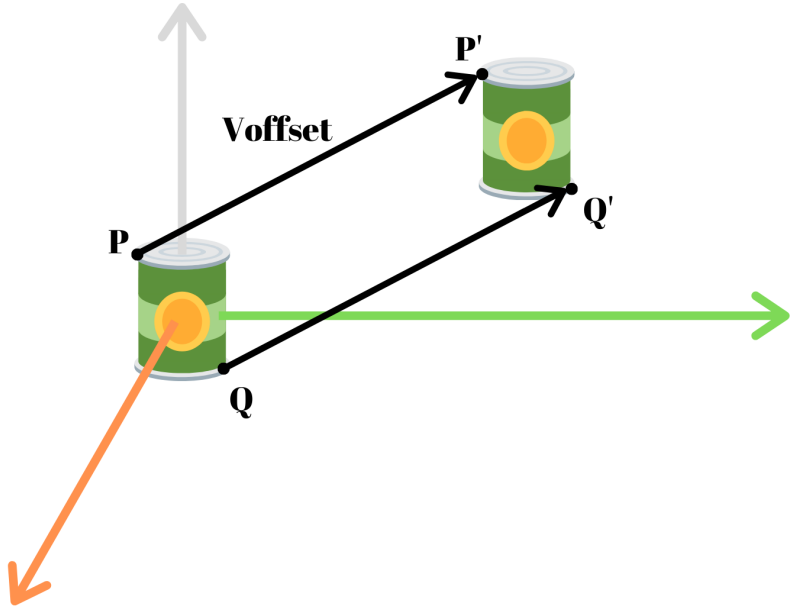


Figure 6.4: An entire object translated from its origin.

6.1.2 Three dimensional rotation

As the name can suggest this type of transformation rotate the object body according to a certain axis. A simple explanation how to rotate can be provided just by writing the point in the space in its polar (2D) case or spherical (3D) coordinate. The rotation around (x, y, z) axis are also called $(roll, pitch, yaw)$ rotation.

$$\begin{aligned} x &= rsin(\theta_z)cos(\theta_x) \\ y &= rsin(\theta_z)sin(\theta_x) \\ z &= rcos(\theta_z) \end{aligned} \tag{6.3}$$

To apply a rotation around z is equivalent to add that angle to the current x angle as shown in Fig. 6.5

$$\begin{aligned} \mathbf{x}' &= rsin(\theta_z)cos(\theta_x + \beta) = rsin(\theta_z)(cos(\theta_x)cos(\beta) - sin(\theta_x)sin(\beta)) \\ \mathbf{y}' &= rsin(\theta_z)sin(\theta_x + \beta) = rsin(\theta_z)(cos(\theta_x)sin(\beta) + sin(\theta_x)cos(\beta)) \\ \mathbf{z}' &= rcos(\theta_z) \end{aligned} \tag{6.4}$$

Equation 6.4 can be rewritten as shown in equation 6.5

$$\begin{aligned} \mathbf{x}' &= rsin(\theta_z)cos(\theta_x)cos(\beta) - rsin(\theta_z)sin(\theta_x)sin(\beta) = \mathbf{x}cos(\beta) - \mathbf{y}sin(\beta) \\ \mathbf{y}' &= rsin(\theta_z)cos(\theta_x)sin(\beta) + rsin(\theta_z)sin(\theta_x)cos(\beta) = \mathbf{x}sin(\beta) + \mathbf{y}cos(\beta) \\ \mathbf{z}' &= \mathbf{z} \end{aligned} \tag{6.5}$$

Equation 6.5 can be expressed in the following matrix form:

$$\vec{\mathbf{V}}' = \underline{\mathbf{Rz}}\vec{\mathbf{V}} = \begin{bmatrix} cos(\beta_z) & -sin(\beta_z) & 0 & 0 \\ sin(\beta_z) & cos(\beta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.6}$$

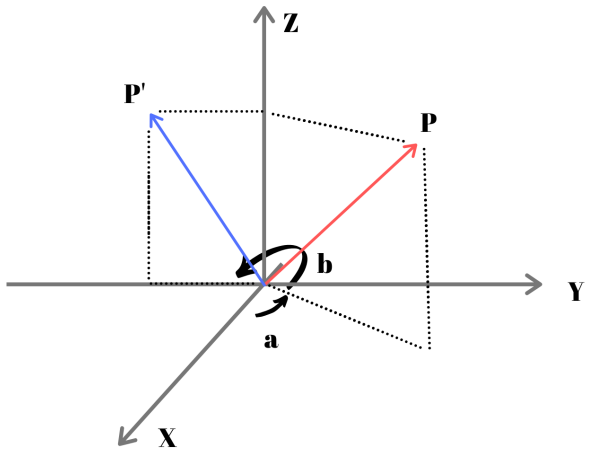


Figure 6.5: 3D rotation of a point around the Z axis

The rotation according to a the other two axis can be performed by multiplying by the following matrices, which can be proved in always in the previous way :

$$\underline{\mathbf{Rx}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\beta_x) & -sin(\beta_x) & 0 \\ 0 & sin(\beta_x) & cos(\beta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \underline{\mathbf{Ry}} = \begin{bmatrix} cos(\beta_y) & 0 & sin(\beta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta_y) & 0 & cos(\beta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

The overall effect to the object body is shown in the figure 6.6.

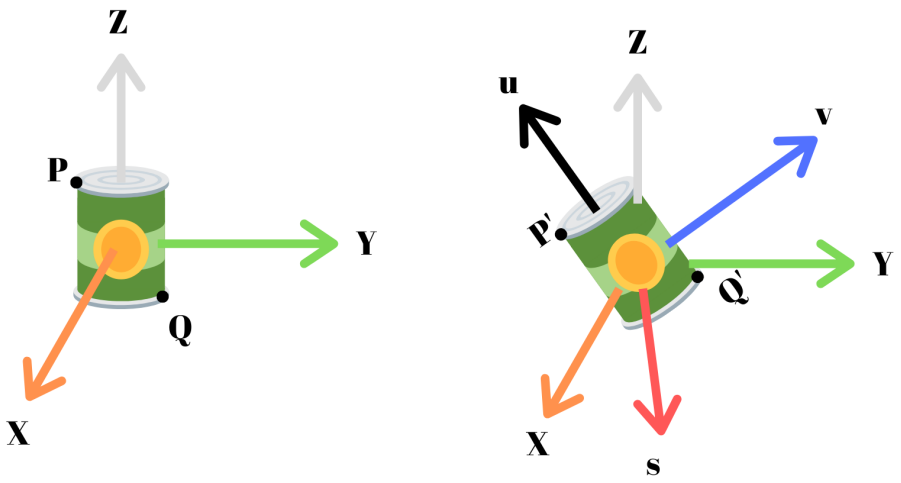


Figure 6.6: Object rotated in the space, showing the difference between the implicit axes to the world ones

6.1.3 Three dimensional scaling

The scaling matrix is the following

$$\vec{\mathbf{V}}' = \underline{\mathbf{Tr}}\vec{\mathbf{V}} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{bmatrix} \tag{6.8}$$

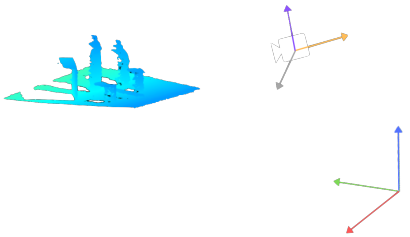
In this project this transformation is not applied,since the size of the object must be preserved

6.1.4 The camera matrix

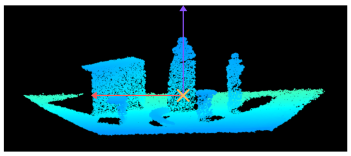
This matrix identify the position and the orientation of the camera in the space respect to the world origin.

$$\underline{\mathbf{C}} = \begin{bmatrix} rx & ry & rz & tx \\ ux & uy & uz & ty \\ wx & wy & wz & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.9}$$

In the matrix camera matrix, equation 6.9, the r,u and w represent the equivalent rotation while the t represents the translation respect the word origin.



(a) Scene, camera and base.



(b) Point of view from the camera.

6.1.5 The projection stage

This stage aims to correct the view of the object, modulating its size according to the its distance respect to the camera. Intrinsically farther is an object smaller it is, but it is also valid for each point that made up the same object.

Exists two main ways of projection :

- In the orthographic the object keeps its aspect ratio regardless the distance from the near plane.

- In the perspective the body is deformed according to its distance from the near plane, equation 6.10.

In the orthographic projection the shape of the frustum is a rectangle, so the far and near plane has the same dimension, while in perspective one the near and far plane have different size, which means that what is in the back is compressed while what is in the front is enlarged. The difference between the two is the shape of the frustum will lead to a two different deformation when the object model is transformed to the normalised coordinate device system, as shown in the figure 6.8 .

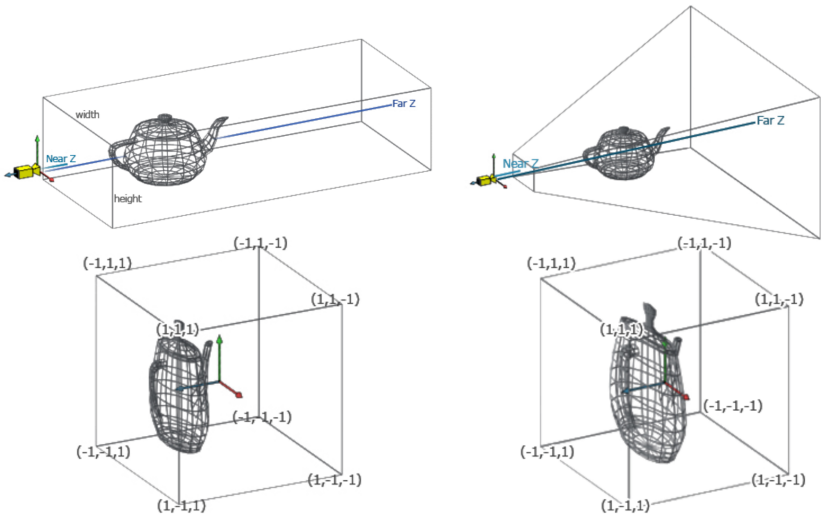


Figure 6.8: Difference in final NDC's object according to the frustum's shape, from [21].

Since the camera has the same type of visual deformation given by the perspective projection, the latter one is the the right one to be used.

$$\underline{\mathbf{P}} = \begin{bmatrix} \frac{2*FX}{Width} & 0 & 1 - \frac{2*CX}{Width} & 0 \\ 0 & \frac{2*FY}{Height} & \frac{2*CY}{Height} - 1 & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2*Far*Near}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{6.10}$$

Since the object model must be projected in the same way that the camera see the world intrinsic parameters of the camera have to be used. Camera intrinsic parameters :

- FX : 542.556
- FY : 537.666
- CX : 325.065
- CY : 224.214

Differently from the transformation described above the projection is not an "affine" transformation so w value can change and consequently a normalisation to go back to homogeneous coordinated is needed.

6.1.6 Back-face culling and clipping

Both of them are common technique in computer graphic to judge if a polygon have to be or not rasterised. The back face culling removes triangles that are not intrinsically visible from the camera prospective, while the clipping removes the faces out of the camera field of view. Both of them are executed when the triangle is transformed in the NDC system.

The back-face culling look at the normal direction of the model face and understand if it points in the same direction to the camera, Fig 6.9. If the face has the same direction means that it is not visible, because covered from other faces pointing towards the camera .

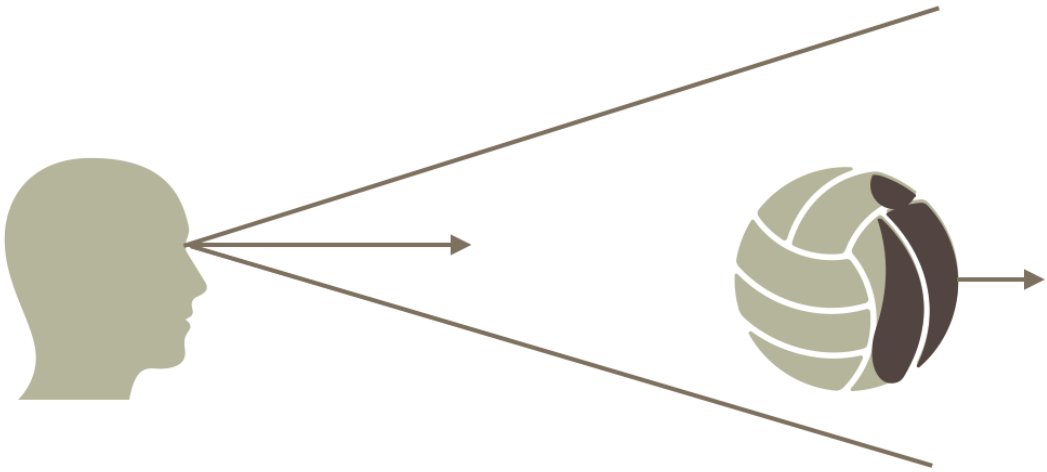


Figure 6.9: Back-face culling

$$\vec{\mathbf{C}}_{direction} \cdot \vec{\mathbf{N}}_{direction} > 0$$
$$\text{C is the camera} \qquad (6.11)$$
$$\text{N is the triangle normal}$$

The clipping is another algorithm is another important algorithm inside the graphic pipeline. It removes the triangles out from the normalised cube, because they are not visible when transformed in the screen space. There are more complex algorithms where instead of removing the triangle, if partially out, they go to interpolate the edge to have a better visual effect.

6.2 Rasterization

This stage aims to fill the triangle, projected in 2D, in the screen space. The vertices of the triangle can be anywhere in the pixel area and the triangle shape can be whatever. A fixed algorithm and rules to define the filling of the triangle are needed to correctly fill the screen space. In literature there are a lot of rasterization algorithms, like the Bresenham specialised to draw lines and optimised to consume very few resources, no floating point calculations,[20]. The idea is to store the derivative value, since it is constant, and have a fixed step of increment, the following code is a very simple version.

```
void DrawLine_BR(int x1,int y1,int x0,int y0){
        // since a line has a fixed derivate
        // we can store per partial results
        dy = x1 - x0;
        dx = y1 - y0;
        //value to judge
        d = -dx + 2*dy;
        //starting coordinate
        x = x0;
        y = y0;
        fill(x, y) ;// fill the starting coordinate
        while x < x1 {
                if (d >= 0) {
                // overcome the mid point, the pixel to draw
                // is in the next line
                 d = d - 2*dx +2*dy;
                 y = y + 1;
                 x = x + 1;
                }
                else {
                 d = d + 2*dy;
                 x = x + 1;
                }
                // each cycle I draw a pixel
                fill(x, y);
        }
}
```

Listing 6.1: Simple Bresenham algorithm

Since the triangle needs to be interpolate and it is defined three lines using Bresenham algorithm is not the most suitable choice. Another very famous and light, in terms of hardware consumption , algorithm to rasterize triangles is the one from proposed by Pineda [22]. The basic idea is to used the direction between two points to create two signed regions, positive and negative, according to the sign of the cross product between a generic point in the relative side side,Fig 6.11. Notice that for Pineda the cross product is also called edge function, equation 6.12 , in which there is a centre, a fixed point and moving point. For a clockwise winding triangle, between the point zero and one, the zero point act like a centre and the one as a fixed point. If P, the generic point, belongs to the edge the output of the edge function is zero, if it is on the right it is positive, while if it is in the left it is negative. Furthermore, in order to not waste operations a smaller box, which contains the triangle, is used to reduce the searching space.

$$Edge_{FC} = (P_x - C_x)(F_y - C_y) - (P_y - C_y)(F_x - C_x) \tag{6.12}$$

C is the centre, F is the fixed point , P is the moving point.

Figure 6.10: Normal direction direction according to the side of the line

By applying this idea among all the points 7 areas are obtained in which the only one inside the triangle have all of the cross products positive, Fig6.12a.

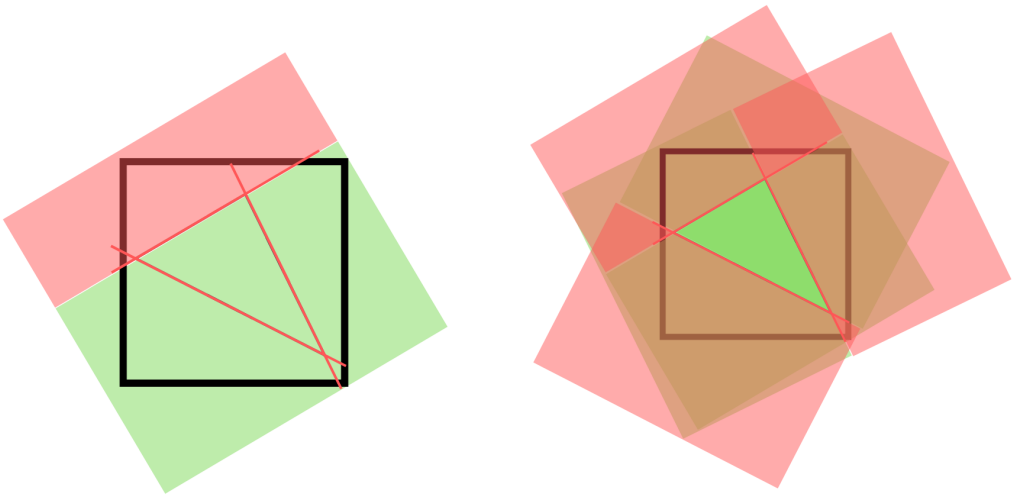


Figure 6.11: Edge function applied to the whole triangle

As in Bresenham, the middle point is the one tested. As shown in figure 6.12b, the pixel grid can be overlapped to the triangle and the centre of the pixel tested. For

(a) Triangle's bounding box area, signed

(b) Pixel grid overlapped to the triangle

a generic point can be defined three edge equations, equation 6.13. The graphical meaning is shown in figure 6.13

$$\begin{aligned} E_{10}(P) &= (P_x - V0_x)(V1_y - V0_y) - (P_y - V0_y)(V1_x - V0_x) \\ E_{21}(P) &= (P_x - V1_x)(V2_y - V1_y) - (P_y - V1_y)(V2_x - V1_x) \\ E_{02}(P) &= (P_x - V2_x)(V0_y - V2_y) - (P_y - V2_y)(V0_x - V2_x) \end{aligned} \tag{6.13}$$

Since the step applied is constant, the value of the next pixel can be constructed by the previous value. First a generic step can be described by the following equation

6.14.

$$\begin{aligned}
&\Delta E_{FC}(P_{i+1} - P_i) = (P_{x_{i+1}} - VC_x)(VF_y - VC_y) + \\
&-(P_{y_{i+1}} - VC_y)(VF_x - VC.x) - (P_{x_i} - VC_x)(VF_y - VC_y) + \\
&+(P_{y_i} - VC_y)(VF_x - VC_x) = \\
&= (P_{x_{i+1}} - P_{x_i})(VF_y - VC_y) - (P_{y_{i+1}} - P_{y_i})(VF_x - VC.x)
\end{aligned} \tag{6.14}$$

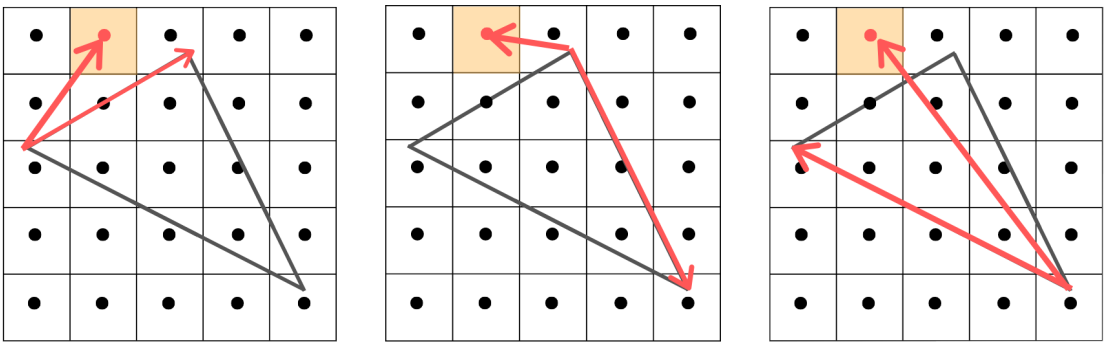


Figure 6.13: Three cross product for a generic point P

Equation 6.14 can decompose by two independent increment of the in the x and y direction, equation 6.15

$$\begin{aligned}
dy_{10} &= (V1_y - V0_y), (P_{y_{i+1}} - P_{y_i}) = 0 \wedge (P_{x_{i+1}} - P_{x_i}) = 1 \\
dx_{10} &= (V1_x - V0_x), (P_{y_{i+1}} - P_{y_i}) = 1 \wedge (P_{x_{i+1}} - P_{x_i}) = 0 \\
dy_{21} &= (V2_y - V1_y) \\
dx_{21} &= (V2_x - V1_x) \\
dy_{02} &= (V0_y - V2_y) \\
dx_{02} &= (V0_x - V2_x)
\end{aligned} \tag{6.15}$$

The decomposition in equation 6.15 allows equation 6.13 to be rewritten as 6.16.

$$\begin{aligned}
E_{10}(P) &= (P_x - V0_x)dy_{10} - (P_y - V0_y)dx_{10} \\
E_{21}(P) &= (P_x - V1_x)dy_{21} - (P_y - V1_y)dx_{21} \\
E_{02}(P) &= (P_x - V2_x)dy_{02} - (P_y - V2_y)dx_{02}
\end{aligned} \tag{6.16}$$

To have a lighter notation equation 6.17 can be substituted to equation 6.16 to

obtain equation 6.18

$$\begin{aligned} dy_{P0} &= (P_y - V0_y) \\ dx_{P0} &= (P_x - V0_x) \\ dy_{P1} &= (P_y - V1_y) \\ dx_{P1} &= (P_x - V1_x) \\ dy_{P2} &= (P_y - V2_y) \\ dx_{P2} &= (P_x - V2_x) \end{aligned} \tag{6.17}$$

In this way each point can be the starting point.

$$\begin{aligned} W_{10}(y_s, x_s) &= E_{10}(P_{start}) = dx_{P_s0}dy_{10} - dy_{P_s0}dx_{10} \\ W_{21}(y_s, x_s) &= E_{21}(P_{start}) = dx_{P_s1}dy_{21} - dy_{P_s1}dx_{21} \\ W_{02}(y_s, x_s) &= E_{02}(P_{start}) = dx_{P_s2}dy_{02} - dy_{P_s2}dx_{02} \end{aligned} \tag{6.18}$$

The main advantage with this approach is that all of the coefficients needs to be evaluated only once and when the rasterization algorithm moves inside the bounding box to test the pixels only additions and subtraction can be used. The three coefficients W_{FC} must be positive to have a pixel inside the triangle, the use of additions or subtractions is defined by the rasterization direction and most of these coefficients can be reused to interpolate the depth between the vertices.

$$\begin{aligned} W_{10}(y_{k+1}, x_k) &= W_{10}(y_k, x_k) - dx_{10} \\ W_{21}(y_{k+1}, x_k) &= W_{21}(y_k, x_k) - dx_{21} \\ W_{02}(y_{k+1}, x_k) &= W_{21}(y_k, x_k) - dx_{02} \\ W_{10}(y_{k-1}, x_k) &= W_{10}(y_k, x_k) + dx_{10} \\ W_{21}(y_{k-1}, x_k) &= W_{21}(y_k, x_k) + dx_{21} \\ W_{02}(y_{k-1}, x_k) &= W_{21}(y_k, x_k) + dx_{02} \end{aligned} \tag{6.19}$$

$$\begin{aligned} W_{10}(y_k, x_{k+1}) &= W_{10}(y_k, x_k) + dy_{10} \\ W_{21}(y_k, x_{k+1}) &= W_{21}(y_k, x_k) + dy_{21} \\ W_{02}(y_k, x_{k+1}) &= W_{21}(y_k, x_k) + dy_{02} \\ W_{10}(y_k, x_{k-1}) &= W_{10}(y_k, x_k) - dy_{10} \\ W_{21}(y_k, x_{k-1}) &= W_{21}(y_k, x_k) - dy_{21} \\ W_{02}(y_k, x_{k-1}) &= W_{21}(y_k, x_k) - dy_{02} \end{aligned} \tag{6.20}$$

Note* : Since all the partial values are evaluated, calculating the normal sign is very easy. From a mathematical point of view the area of the triangle is equal to the length and direction of the output vector. Since the area will be also useful later for the interpolation of the depth, necessary for equation 6.21 can be evaluated in the following way :

$$2Area_{face} = (V2_x - V0_x)(V1_y - V0_y) - (V2_y - V0_y)(V1_x - V0_x).$$

(a) Pixel analysed.

(b) Pixel rasterized.

That can be rewritten in the following form "recycling" the previous calculations:

$$2Area = (dy_{02} * dx_{10}) - (dx_{02} * dy_{10})$$

By looking at the sign is possible to understand if the face have to be or not rasterised, since in the transformation chain each point is also multiplied by the camera matrix. If the sign is lower or equal to zero the triangle is not visible.

6.2.1 Depth Interpolation

Each triangle is described by three tuple of points. From the latter an interpolation, to all of the points in between is needed, to correctly shade the object. The most common way to interpolate the depth of a triangle is by using the barycentric coordinate[23], used also to shade colours. The barycentric coordinate is a type of homogeneous coordinate system defined by the points of the triangle. The main idea is to weight the three information, coming from the vertices, according to the distance with each point.

$$z_p = \lambda_0 z_{V_0} + \lambda_1 z_{V_1} + \lambda_2 z_{V_2}$$

$$\lambda_i = \frac{Area_i}{Area_{Tot}} \tag{6.21}$$

$$\lambda_0 + \lambda_1 + \lambda_2 = 1$$

The coefficients λ_i inside the triangle are lower or equal to 1 and their value change according to the distance. For example if the point that is under evaluation is in one vertex, the correspondent λ_j will be one and the other two will be zero. In an analogue way if the point is lying in the edge a certain vertex does not touch λ_j becomes zero. Intuitively the depth weight is directly related to the importance of the portion of area contained, figure 6.15.

Since all lambda coefficients defined in 6.21 are the relative area over the overall, by using equation 6.18, the same equation can be reinterpreted as shown in equation

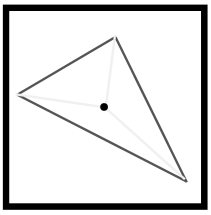


Figure 6.15: Barycentric coordinates

6.22.

$$
\begin{aligned}
2Area_{01}(P) &= W_{01}(P) \\
2Area_{12}(P) &= W_{12}(P) \\
2Area_{20}(P) &= W_{20}(P)
\end{aligned}
\tag{6.22}
$$

$$z_p = \frac{W_{10}z_{V_0} + W_{12}z_{V_1} + W_{20}z_{V_2}}{2Area_{face}}$$

$$z_p = \frac{Area_{01}(P)z_{V_0} + Area_{12}(P)z_{V_1} + area_{20}(P)z_{V_2}}{Area_{face}}$$

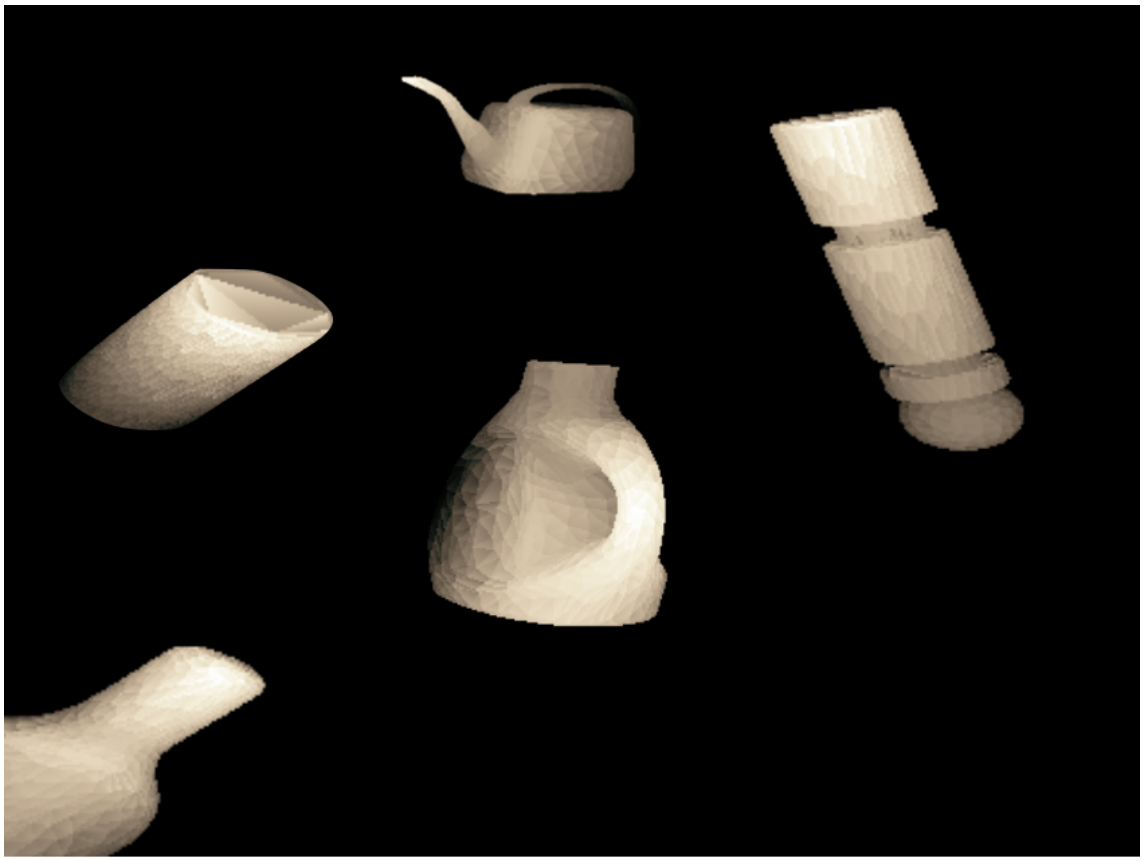


Figure 6.16: Example took from the algorithm implemented

Part III

Thesis implementation and results

Chapter 7

Hardware Implementation

In this chapter the complete hardware architecture will be proposed and described. **Note*** in this project the $HEATMAP_{size}$ is equal to 6300, formed by the sum of the three level of heatmap and $NUM_{particles}$ is equal to 510.

7.1 Essential memories

To maximise the speed and reduce the power consumption the number of external memory request was minimise and the delay "hidden" between possible operations.

The information needed to start the second stage are the following :

- **Depth Scene** (480x640) : It is in a floating point format, due to the camera standard. The depth was clipped to 4 meters keeping the maximum resolution of the camera , 1 mm, and stored in a BRAM of the size of 12 bits : 2 bit of int and 10 bit of fractional part.
- **Raw Heatmap** (26* $HEATMAP_{size}$) : This is the first stage raw heatmap, which will be converted directly to the final one without storing it. The number 26 is composed by 21 probabilities, 4 raw bounding boxes and one confidence score. Image 7.2b shows how for each "pixel" of a generic level of the pyramid CNN the information are distributed.

The post-processing module will produce :

- **probability** (21* $HEATMAP_{size}$) : It contains the probability distribution after the post-process module. The number of bit for the integer part is 0, while 8 bit are used to describe the fractional part. This truncation shows enough precision for the calculation.
- **bounding box** ($HEATMAP_{size}$) : It contains (x, y) of the centre, 10 bit due to the image size and $(width, height)$ relative for that specific bounding

box, 8 bit due to clipping. These informations are often used together, so in order to reduce the memory access and compact the informations they can be merged in a single information of 18 bit.

Aside the original informations other arrays are needed :

- The current **object model** is stored in a file with the extension .obj contain the 3D models of the possible object inside the scene. These rendered objects are obtained with the same camera and after downsampled. Each file contains a header, which gives information about the number of vertices and faces. Is also important to point out that only the vertices with "v" will be used, not in vn, which is the vertex normal in (x,y,z) form. Vertices are described in (x, y, z) coordinate and the information are written in the same order. Faces instead provide information on the vertices' connections. A face is made of three vertices, like a triangle for FEM modelling. $v1//v1$ $v2//v2$ $v3//v3$. The object model is stored in the following way inside the FPGA :
  - 3 array containing the information about each geometric vertices : 3 bit of integer part and 10 of fractional one. This passage from floating point to fixed means that the object model is also quantized.
  - 1 array of 30 bits to contain the information about the faces. Since the model used is a downsampled one the maximum number of vertices is bounded to 1024, which means that we need only 10bit to correctly address them. With this packing the number of request to the face index array is reduced.

```
switch (case_select) {
    case 0:
      Vert_address = Tri_Mem[ext_select];
      Vert.x = V_Mem[0][Vert_address.range(9, 0)];
      Vert.y = V_Mem[1][Vert_address.range(9, 0)];
      Vert.z = V_Mem[2][Vert_address.range(9, 0)];
      case_select = 1;
      break;
    case 1:
      Vert.x = V_Mem[0][Vert_address.range(19, 10)];
      Vert.y = V_Mem[1][Vert_address.range(19, 10)];
      Vert.z = V_Mem[2][Vert_address.range(19, 10)];
      case_select = 2;
      break;
    case 2:
      Vert.x = V_Mem[0][Vert_address.range(29, 20)];
      Vert.y = V_Mem[1][Vert_address.range(29, 20)];
      Vert.z = V_Mem[2][Vert_address.range(29, 20)];
      case_select = 0;
```

```
20          break;
```

Listing 7.1: Model memory addressing

- **Sample Memory** ($2\text{x}NUM_{particles}$) : The need of two sample memory is due to the dataflow pragma and the implementation of the diffusion stage. Broadly speaking, in each iteration one of the two sample memory is select as input and the other as output memory. Each sample element is composed by :
  - **Three coordinate in the space** (x, y, z) : 3 bit of integer part and 10 of fractional one.
  - **Three angles** $(roll, pitch, yaw)$: 4 bit of integer part and 8 of fractional one.
  - **Absolute index**: it connects a sample to a specific bounding box in the array, 13 bits are needed to address up to 6300.
- **Heuristics Memory** (6300): It must describe a volume in the space respect to the base of the robot. The information needed are the following :
  - (x_{min}, x_{max}) : 3 bit of integer part and 10 of fractional one.
  - (y_{min}, y_{max}) : 3 bit of integer part and 10 of fractional one.
  - (z_{min}, z_{max}) : 3 bit of integer part and 10 of fractional one.

A quick review can found in the table 7.1

Due to dataflow constrains a second sample memory and relative index memory are needed, this memory overhead allows to reduce the data movements and allows the sample memory to be treat like a ping-pong memory across two different iterations. The introduction of the concept of relative and absolute index reduce the data movement. For example, in the sorting instead of moving all the characteristics of a sample, the sorting module can just sort a memory of relative indices. The latter during the resampling is just read a used to pick the original information, transform and diffuse it in the output sample memory.

7.1.1 Fixed point

Differently from the GPU and base version this project is transformed to work in fixed point representation, because the DSP inside the FPGA are optimised for fixed point calculation and more in general because they consume less power[24]. The fixed point representation is usually expressed in the format $QI.F$ where I stands for the number of integer bits and the F for the fractional ones. Anyway, it can be interpreted just as binary number in which an imaginary dot is placed after

	Size	xyz_Data_T	bit	angle_Data_T	bit
Sample Memory	2x510	x_S, y_S, z_S	13	$roll_S, pitch_S, yaw_S$	12

	Size	xyz_Data_T	bit
Heuristic Memory	6300	$x_{min}, y_{min}, z_{min}$ $x_{max}, y_{max}, z_{max}$	13

	Size	BB_Data_T	bit
Bounding Box Memory	6300	$x_c + width_i$ $y_c + height$ $h_{indx} + w_{indx}$	18

	Size	Prob_Data_T	bit
Probability	21x6300	$probability_i$	8

	Size	Abs_Indx_Data_T	bit
Absolute Index Memory	2x510	$index_i$	13

	Size	Depth_Data_T	bit
Depth Memory	480x680	$pixel_i$	12

Table 7.1: Memory table

I bits. For this reason operation like the division or multiplication for power of two can be reduced to dot shifting or expansion.

In this thesis the convention that will be used is the following

- **uint<N>** : N bits of integer part
- **int<N>** : N-1 bits of integer part and 1 bit of sign
- **ufixed<N,I>** : N bits of total bits and I bits of integer part
- **fixed<N,I>** : N bits of total bits , I-1 bits of sign and 1 bit of sign

7.1.2 Design Flow

Since this project was developed through the tools provided by Vivado the following design flow was used :

- Organise the ideas and understand how to optimise a module by evaluating its pros and cons.

- Generate a possible pseudo-code to validate the solution and the timing to accomplish the result.
- Write the equivalent code to represent the wanted .algorithm with HLS and generate the synthesised version
- Test-bench the hardware and validate the design with a simulation. If a fail occurs by checking the code, the pseudo one and the timing generated act properly.
- Export the IP core, create a Vivado project, program the micro-controller and generate the bit-stream.
- Validate the design. If the it is not a good design restart from the beginning with the acquired knowledge.

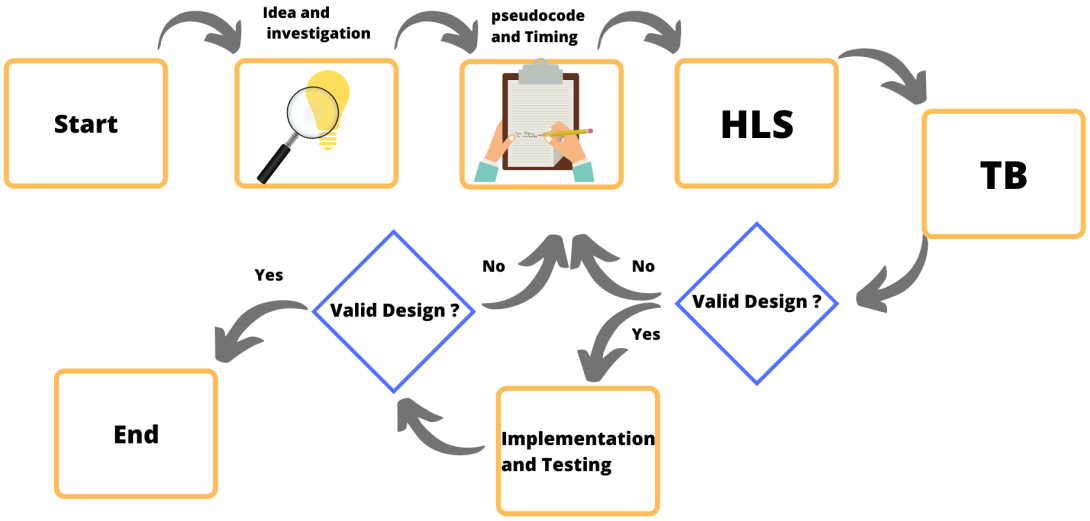


Figure 7.1

7.2 Neural network

To execute the first stage, the convolutional neural network, the Deep neural network Development kit, DNNDK [25], provided by Xilinx® is used. This kit provides a series of functionalities, as quantization, optimisation and compression for the most common structure of CNN. The specific module, which is designed to accelerate in hardware the CNN, is called DPU[26], deep learning processor unit. The latter is highly customizzable because it can allows classification, segmentation, detection and tracking of the various object's classes specified by the input. To map the NN algorithm Xilinx® provides a compiler called DNNC, deep neural network compiler, to maximise the performance of the algorithm for the DPU.

The tool-chain provided by the DNNDK follows these steps :

1. Compression : This stage, executed by the DECENT, Deep Compression Tool, consist to reduce the number of parameters for the network and consequently re-tune them. After, the network's parameters are quantized from the usual 32 bit floating point to 8 bit integer, transforming the overall NN from a floating to a fixed one. The main reason is that in this way the network itself reduce the computational cost, increase the power efficiency and it's faster. Notice that also the output activation functions will be fixed point.

2. Compiling : executed by the DNNC, deep neural network compiler, transform the reduced neural network in to an intermediate IR, instruction set register, and the latter will be optimised to the DPU module.

3. Create the API

4. Compile the API for DPU IP core

5. Run the DPU module

7.2.1 DPU MODULE

As said before the DPU, is a module very customizable, capable to execute the more common CNN as VGG, ResNet, YOLO, etc. The configurable options are

- The number of DPUs : instantiate up to three DPU core inside one IP reaching higher performances but consuming more area.

- The type of Arch : this parameter vary the type of convolutional engine inside the architecture allowing to choose the pixel and channel parallelism.

- The type of clock mode for the AXI input connection : common or independent.

- The maximum number of DSP48 in cascade, higher means less logic and worse timing. The quote comes from PG338 v1.2 March 26,2019 by Xilinx®

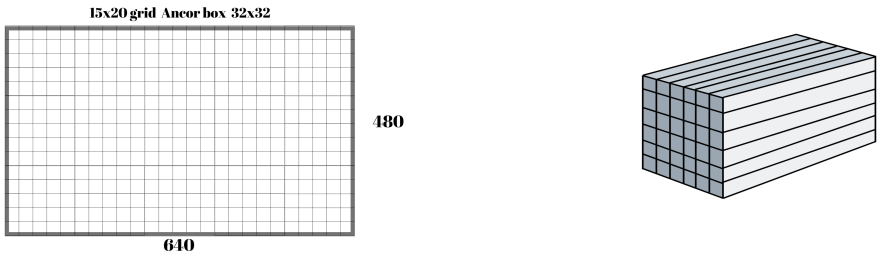
  > Xilinx® recommends selecting the mid-value, which is 4, in the first iteration and adjust the value if the timing is not met.

- DSP usage : low or high and changes according to the arch of the DPU.
- Ultra-RAM usage

The module itself just compute the network and cannot work alone, so it needs a application processing unit, as a micro-controller, that serve interrupts and mange the data transfer. The hardware architecture is similar to a micro code architecture. The instructions, generated from the DNNDK, are loaded in the module and read.

7.3 Post-process

In version of the neural network the heatmap used has three layers or levels, the first 15x20 , the second 30x40 and the third 60x80, to produce an overall size of 6300. Due to the output format, described in subsection 7.1, means that it cannot be used directly to generate a bi-dimensional probability distribution and furthermore the real bounding boxes must be evaluated from the raw parameters and the anchor box of the original image, image 7.2a. The architecture developed is shown in the figure 7.3.



(a) Each level of heatmap has an ancorbox : a fixed grid to the original image.

(b) First stage output graphical representation.

Figure 7.2

Since the algorithm to convert the first stage output to a probability distribution needs the use of exponential, quite expensive in hardware, the design uses , to reduce the computational delay of this stage, a ROM to store the exponential function

values. The implementation is possible because the input is bounded to 8 bit, 256 cells with a range that goes from $1.12 * 10^{-}7$ up to $8.8 * 10^{6}$.

At the beginnin of the post-process module, the "splitter" module reads the input probability, pick from the memory the corresponding exponential value and transmit it into an accumulator and into a delay buffer.

$$Confidence_{score} = 1 + e^{c} = 1 + ROM_{exp}(c) \tag{7.1}$$

$$Probability_{j} = \frac{e^{p_j}}{\sum_{i=1}^{21} e^{p_i}(1 + e^{c})} \tag{7.2}$$

Once the overall sum is evaluated is possible to execute the soft-max. The latter

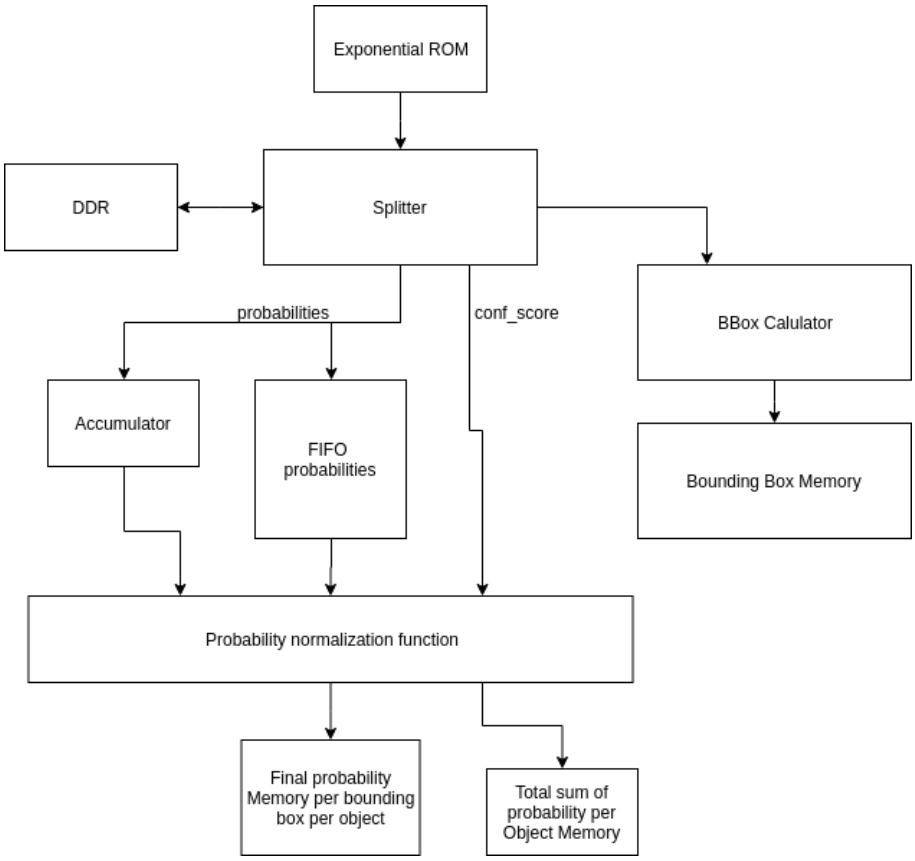


Figure 7.3: Post process to extract probability heatmap and bounding boxes informations.

is necessary to find the normalise probabilities across all the object ones, per each bounding box. Instead of executing 21 division a single reciprocal module is added after the accumulator stage, figure 7.4. The reciprocal of the overall sum will speed

up the normalisation process since the division is substituted by a multiplication, faster. To archive the correct timing a FIFO is added in parallel to the accumulator and reciprocal module according to the delay of both modules. This approach, used also in other modules, lets the architecture to archive a good speed up during any normalisation processes. Moreover, since the next macro module, Initialisation module, needs the new sum of probabilities, it is evaluated concurrently to the output probability. In the same way also the final bounding boxes are composed and clipped to a maximum size 200x200.

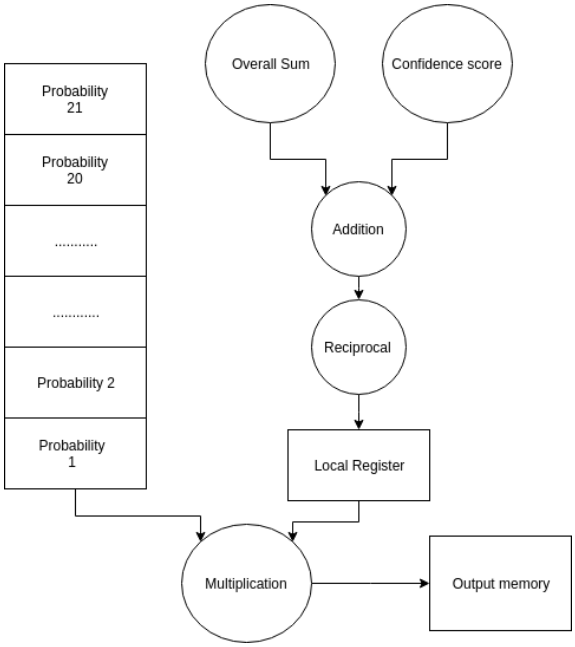


Figure 7.4: Zoom in to the normalisation algorithm.

7.4 Initialisation

The initialisation module generates the random samples used inside the iterative process, the particle filter. The proposed architecture is shown in figure 7.5.

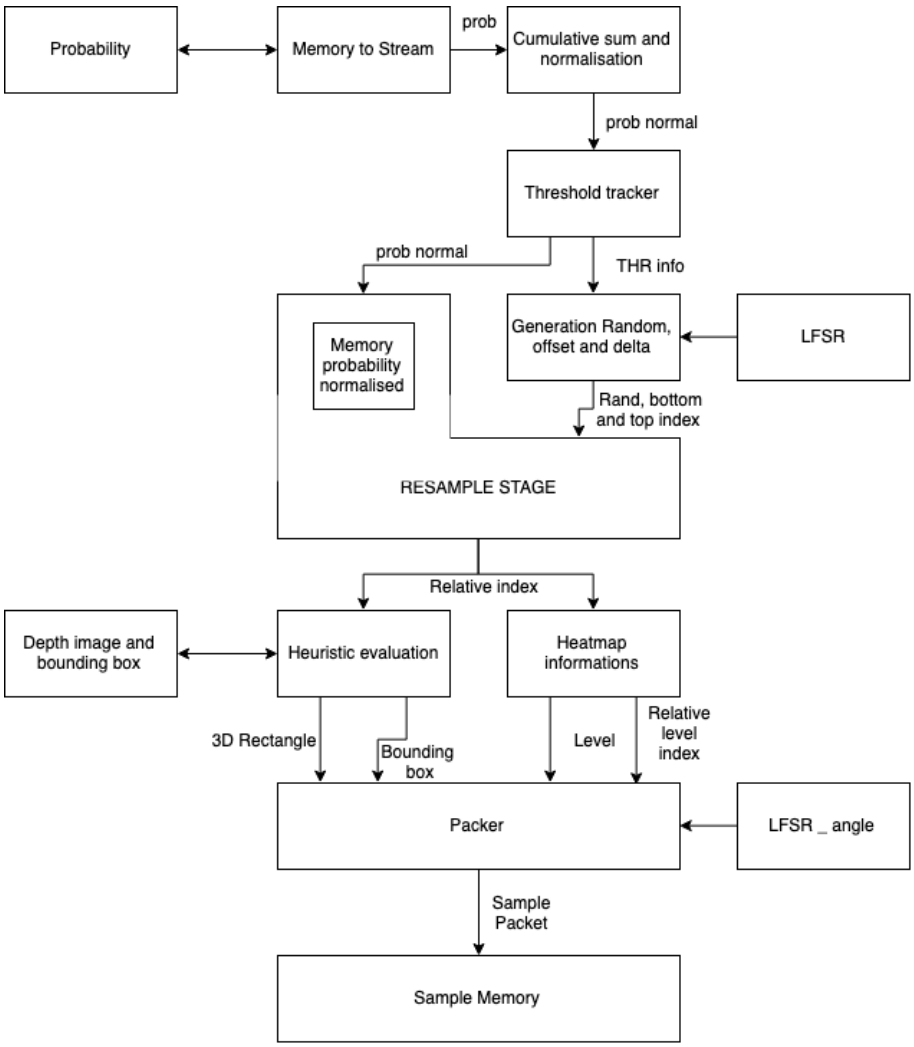


Figure 7.5: Architecture proposed for the sample initialisation.

The architecture can be divided in three main parallel processes :

- The random number generators : It has the purpose to produce as output random angles between $\pm\pi$ and the random values used for the sampling.
- The generation chain : It evaluate and select according to the first stage weight the most likely absolute index , or equivalently bounding boxes.

- The Heuristic generator : It fills the heuristic memory from the depth image and transmit the relative heuristic information to generate the sample in the correct space location.

The module has two inputs :

- The bi-dimensional distribution extracted from the first stage by the post-process module .
- The overall sum of the probabilities of the heatmap is used to evaluated the normalisation coefficient to calculate the cumulative normalised distribution.

Once the Cumulative sum and normalisation module will start producing, in a sequential way, the output values the Threshold tracker module will analyse them and re-transmitted to a temporary memory. The BRAM stores the cumulative normalised density function used by the sampling stage. Once the threshold tracking finished to analyse the normalised values it will pass the information collected to the bound generator. The latter will use the data received to produce custom addresses, according to the random number used. The sampling module module receives each iteration from the latter module two addresses and a random number. Once read the temporary array is scanned to find the right value and the correspondent index used to pick the bounding box and the heuristic. Now that the area is selected a sample is constructed from the heuristic informations and stored in the "starting" memory.

Each sub-block is described in the following list:

1. **Evaluation of the normalised cumulative sum**

$$w_{n_i} = \frac{w_i + \sum_{q=1}^{i-1} w_q}{\sum_{k=1}^{N} w_k} \quad , \; N \text{ number of probabilities from the first stage} \tag{7.3}$$

This module is designed in a fully parametric way with full precision inner calculations. The output is truncated, to reduce the size of the storing memory, according to the precision wanted. In the current version the number of bit kept are 18. This module as like the post process does not store any information, aside the normalisation coefficient and the full resolution cumulative sum. Furthermore, an alternative version of this module was developed previously, in which the overall sum of probabilities is calculated by the module, by reading twice the probability memory. The architecture proposed and its timing are shown respectively in figure 7.7 and 7.6.

By giving a general consideration, since the module is designed in a parametric way also the bit-width for all the local variables are calculated in the same

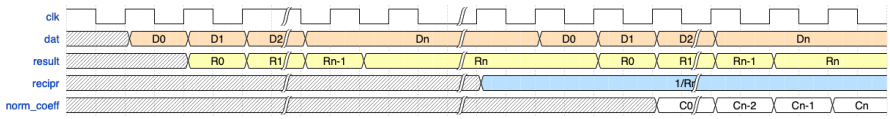


Figure 7.6: Timing Cumulative Sum and normalisation.

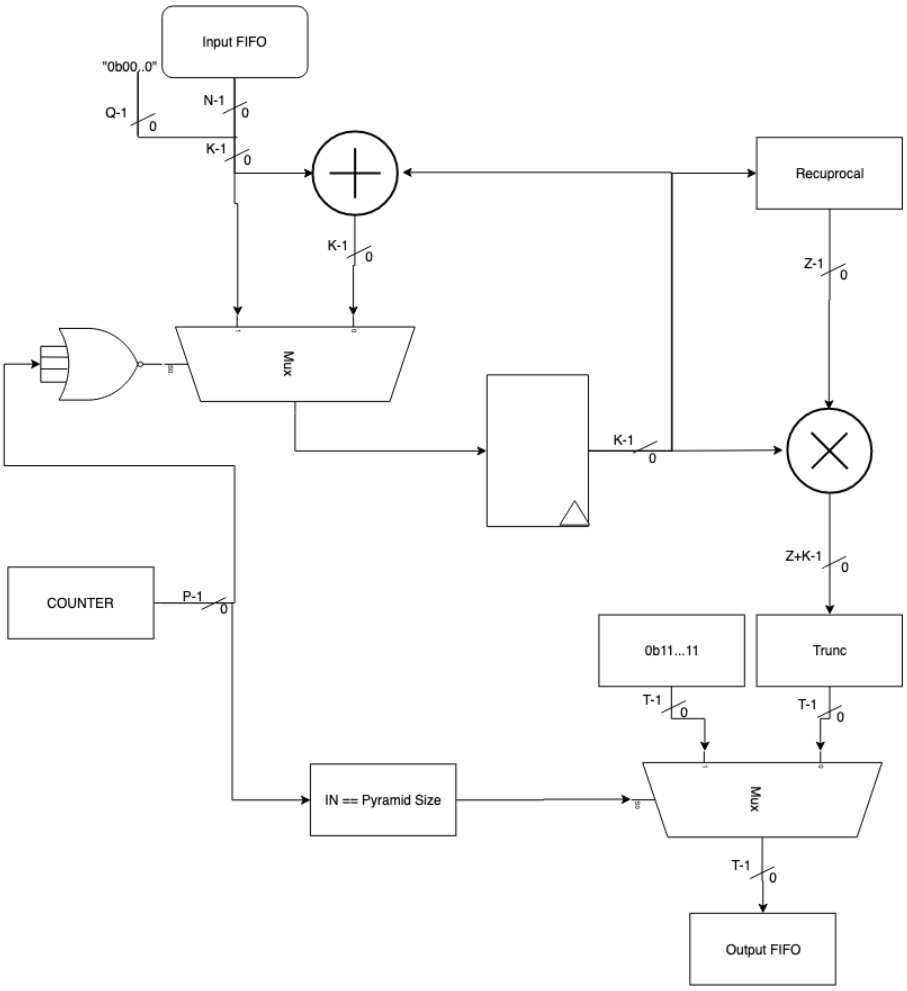


Figure 7.7: Datapath Cumulative Sum and normalisation.

way. The probabilities of the heatmap are unsigned and strictly lower than one, they can be represented as fixed point with ufixed<I,Q> = ufixed<N,0> zero bit for integer part and N for fractional part. The full precision sum bit size depends on the full size of the heatmap, because this parameter set the

number of addition.

$$PyramidSize = \sum_{q=0}^{N_{levels}-1} Height_q * Width_q$$

The number of bit in a sum growth with the $K = ceil(log_2(PyramidSize))$ and it is possible to state that the overall sum fits in ufixed<K+N,K>. Once this value is found is possible to calculate its normalised value by multiply for the normalisation coefficient, the final bit width will be $K + N + Z$. The final output can be reduced to ufixed<T,0>, stricty lower than 1. To compensate the finite precision of the reciprocal module the last element of the sum is saturated to the maximum representable. **Note*** : the reciprocal bit size is not defined, because ,although the reciprocal of fractional part has a fixed dimension

$$\frac{1}{2^{-N}} = 2^N$$

The reciprocal of the integer part has no boundaries, for example $\frac{1}{3} = 0.\overline{3}$ needs infinite precision. According to [12] a practical solution to this problem is to take into account the maximum integer and approximate it.

$$\frac{1}{2^K - 2^{-N}} \approx \frac{1}{2^K} = 2^{-K}$$

This solution of course has its limitations, precision-speaking, but allows to not increase the number of bit from the sum stage and get all the range. Another possible solution is to look at the actual value of the sum, since the max and min values are known, and create different paths. During the experiment was discovered that for our calculations two bit extra for the reciprocal value is necessary.

In our case the input is 8 bit for the fractional part and the overall size is 6300. As consequence the overall sum can be represented in ufixed<13+8,8>, the normalisation coefficient in ufixed<13+8+2,13> and the full precision final multiplication needs ufixed<31,0>. After the truncation the result will be ufixed<18,0> and it is transmitted to the next module, 18 bits are kept in order to have a satisfactory resolution during the sampling and use all the possible bits in a 18K BRAM, which number will be $ceil(PyramidSize/1024) = 5$. The full code can to synthesised the circuit can be found in appendixC.

2. **Threshold tracker**

To select a random bounding box, the sampling stage wastes a lot of time because it scan from the zero cell up to the Q^{th} to pick the index of the first

value, of normalised probability , which overcome the random value, generated by a LFSR. The threshold tracker module is a new stage respect the GPU implementation. The purpose of this module is to reduce number of reading, as a consequence both the power consumption of the sampling stage and the search time for the new sample. To reduce the re-scan problem, the module stores values and indices at specific interval, to minimise the overall number of reading to the current distribution. The step during the default operation is kept constant, unless the difference between two adjacent values exceeds the fixed distance. In the non standard condition, the module create in an adaptive way the next thresholds value to overcame.

The threshold analysis occurs concurrently to the output of the cumulative normalised sum, the proposed architecture and the timing are shown in figure 7.8 and 7.9.

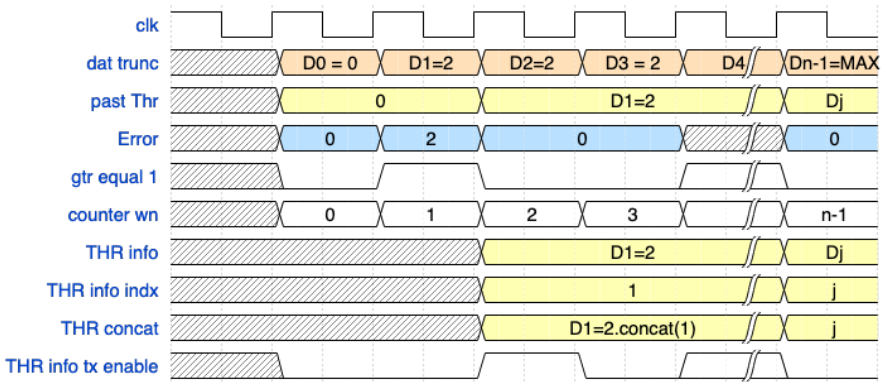


Figure 7.8: Timing Threshold tracker .

The threshold tracker outputs in the bound generator the value concatenated to the index.

$$v_{thr}[Q] = THR(w_n[N]) \ , \ Q \text{ number of threshold stored}$$

At the beginning 2^P thresholds are supposed to be filled with an inner step of 2^{-P}. The counter that keep the information about the probabilities normalised have L bit, while the truncated version of the probabilities have P bits. The threshold information,to be transmitted, is packet as following :

$$Thr_{information}.range(L+P-1, L) = Value_{Trunc}$$

$$Thr_{information}.range(L-1{,}0) = counter_{Wn}$$

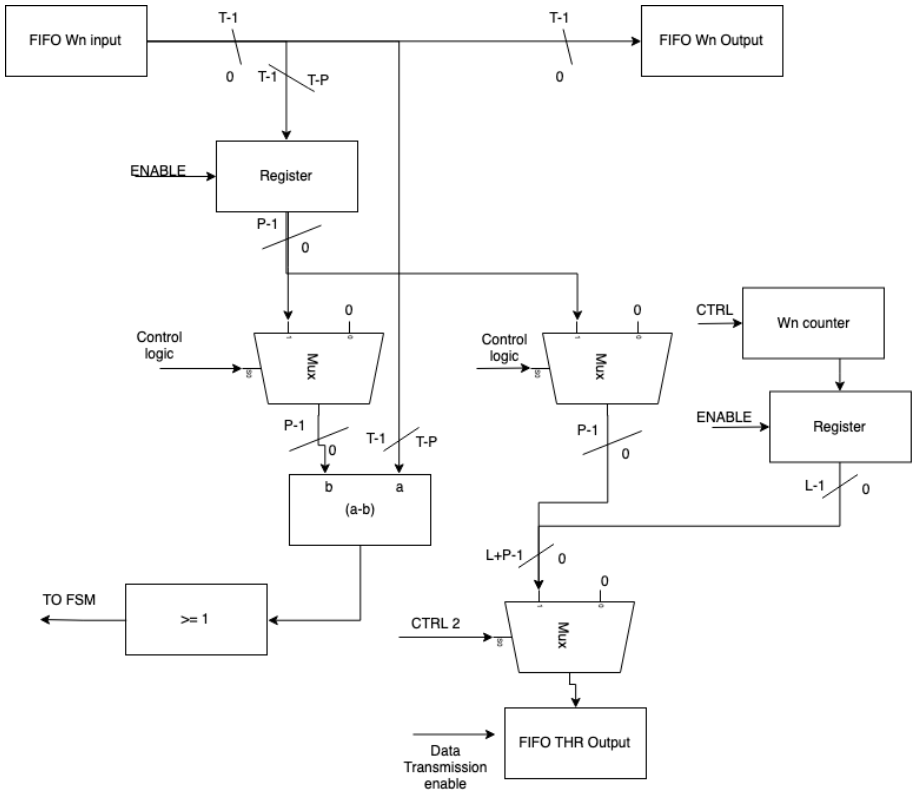


Figure 7.9: Datapath Threshold tracker.

Since all the modules are data driven a fixed number of transmission is needed. If the number of useful threshold is lower than the maximum the module outputs a fixed value, all zeros, to the bound generator to complete the transmission. The code used to synthesise the hardware implementation is the following :

```
void Thr_Analizer(stream<Norm_weights_Data_T> & Norm_weights,
    stream<Norm_weights_Data_T> & Norm_weights_out,
    stream<THR_INFO_Data_T> & THR_INFO) {

  THR_Data_T past_thr = 0;
  THR_P_Data_T Sent_cnt = 0;
  Init_wr_weights: for (u13_Data_T norm_ind = 0; norm_ind <
   PYRAMID_PIXELS;
      norm_ind++) {
#pragma HLS PIPELINE II=1
    Norm_weights_Data_T weights2nsgen_fill;

    // Just copy temporary the information
    Norm_weights.read(weights2nsgen_fill);
```

```
    Norm_weights_out.write(weights2nsgen_fill);

    THR_Data_T curr_thr = weights2nsgen_fill.range(
   BIT_WEIGHT_NORM - 1,
    BIT_WEIGHT_NORM - BIT_THR);
    if (curr_thr - past_thr >= 1) {

      THR_INFO_Data_T THR_INFO_fill = curr_thr.concat(norm_ind)
   ;
      THR_INFO.write(THR_INFO_fill);
      past_thr = curr_thr;
      Sent_cnt++;
    }

  }

  for (; Sent_cnt < SIZE_ARR_THR; Sent_cnt++) {
#pragma HLS PIPELINE II=1
#pragma HLS LOOP_TRIPCOUNT min=1 max=32 avg=16
    THR_INFO.write(0);
  }

}
```

Listing 7.2: BoundGenerator example

3. **Generation of the sampling bounds**

   The module has the task to select the right starting and ending indices according to the random value. All the informations will be transmitted to the sampling stage. To not waste time between two samples, the module works in parallel to the sampling memory reading end evaluate the boundaries. The code used to synthesise the hardware is the following :

```

void Init_gen_rand_bound(stream<THR_INFO_Data_T> & THR_INFO,
    stream<u18_Data_T> & RAND_stream,
    stream<BOUNDS_RAND_Data_T> & Bounds_stream) {

  // CURRENT THR concat to the address  ( 5 + 13 )
  THR_INFO_Data_T thr_array[SIZE_ARR_THR];
#pragma HLS ARRAY_PARTITION variable=thr_array cyclic factor=2
   dim=1

  THR_P_Data_T zero_thr_counter;
  for (THR_P_Data_T counter = 0; counter < SIZE_ARR_THR;
   counter++) {
#pragma HLS PIPELINE II=1
    THR_INFO_Data_T THR_INFO_fill;
```

```
14      THR_INFO.read(THR_INFO_fill);
15      thr_array[counter] = THR_INFO_fill;
16      // THE FIRST THR CAN HAPPEN IN THE ZERO POSITION BUT NOT
        THE ZERO THR
17      if (!counter.or_reduce()) {
18        zero_thr_counter = 0;
19      } else if (!THR_INFO_fill.or_reduce()) { // THIS IS THE
        CONDITION TO MARK
20        zero_thr_counter++;
21      }
22
23    }
24    Init_new_sampl_generation: for (u10_Data_T i = 0; i <
       NUM_PARTICLES; i++) {
25  #pragma HLS PIPELINE II=1
26
27      u18_Data_T rand_numb_fill;
28      u13_Data_T bottom_stream_fill;
29      u13_Data_T top_stream_fill;
30      rand_numb_fill = RAND_stream.read();
31
32      THR_Data_T thr_local = rand_numb_fill.range(BIT_WEIGHT_NORM
        - 1,
33      BIT_WEIGHT_NORM - BIT_THR);
34      if (!thr_local.or_reduce()) { // this fix the lower
        intervall
35        bottom_stream_fill = 0;
36        top_stream_fill = thr_array[0].range(BIT_SUM_INIT - 1, 0)
        ;
37      } else if (thr_local == 31) { // this fix the upper
        intervall
38        bottom_stream_fill =
39            thr_array[SIZE_ARR_THR - 1 - zero_thr_counter].range(
40            BIT_SUM_INIT - 1, 0); //from 31 downto
41        top_stream_fill = PYRAMID_PIXELS - 1;
42      } else {
43
44        Init_gen_rand_bound_trk: for (THR_P_Data_T counter = 2;
45            counter < SIZE_ARR_THR; counter++) {
46  #pragma HLS UNROLL
47          THR_INFO_Data_T temporaneo_Prec = thr_array[counter -
        1];
48          THR_INFO_Data_T temporaneo_act = thr_array[counter];
49          if (thr_local < temporaneo_act.range(BIT_THR +
        BIT_SUM_INIT - 1,
50          BIT_SUM_INIT)) {
51            bottom_stream_fill = temporaneo_Prec.range(
52            BIT_SUM_INIT - 1, 0);
53            top_stream_fill = temporaneo_act.range(
```

```
54            BIT_SUM_INIT - 1, 0);
55            break;
56          } else if (counter == SIZE_ARR_THR - zero_thr_counter -
        1) {
57            break;
58          }
59
60        }
61      }
62      BOUNDS_RAND_Data_T FIll;
63      FIll.range(BIT_RANDOM_NUMBER + BIT_SUM_INIT + BIT_SUM_INIT
       - 1,
64      BIT_RANDOM_NUMBER + BIT_SUM_INIT) = top_stream_fill;
65      FIll.range(BIT_RANDOM_NUMBER + BIT_SUM_INIT - 1,
       BIT_RANDOM_NUMBER) =
66          bottom_stream_fill;
67      FIll.range(BIT_RANDOM_NUMBER - 1, 0) = rand_numb_fill;
68      Bounds_stream.write(FIll);
69    }
70
71  }
```

Listing 7.3: Bound Generator

As a first step the module loads all the Thresholds, after it starts iteratively to load from the LFSR a random value, looking for the relative position inside the threshold array and transmit all the informations .

4. **Sampling stage**

   The module has the purpose to sample from the current distribution. Since the input CDF is bounded to 1, the random number generator output can also be bounded to 1. The way to pick a sample index consist in the following mathematical description.

$$NewSample_i = w_{n_i} | \left[(w_{n_i} \geq U_i) \wedge (w_{n_i} < w_{n_j}, i < j) \right]$$

U_i Uniform random number

   The new sample is the first element that has a normalised probability higher than the random number. The code used to synthesise the hardware is the following :

```
1  void Init_gen_sampleindx(stream<Norm_weights_Data_T> &
       Norm_weights,
2      stream<BOUNDS_RAND_Data_T> & Bounds_stream,
3      stream<u13_Data_T> & sample_indx_stream,
4      Norm_weights_Data_T acc_norm_weights_[6300]) {
5
```

```
6   u10_Data_T saver_counter = 0;
7   Init_wr_weights: for (u13_Data_T norm_ind = 0; norm_ind <
     PYRAMID_PIXELS;
8       norm_ind++) {
9 #pragma HLS PIPELINE II=1
10    Norm_weights_Data_T Norm_weights_fill;
11    Norm_weights.read(Norm_weights_fill);
12    acc_norm_weights_[norm_ind] = Norm_weights_fill;
13  }
14
15  Init_new_sampl_generation: for (u10_Data_T i = 0; i <
     NUM_PARTICLES; i++) {
16    BOUNDS_RAND_Data_T Bounds_stream_fill;
17    Bounds_stream.read(Bounds_stream_fill);
18
19    ap_uint<BIT_SUM_INIT> TOP = Bounds_stream_fill.range(
20    BIT_RANDOM_NUMBER + BIT_SUM_INIT + BIT_SUM_INIT - 1,
21    BIT_RANDOM_NUMBER + BIT_SUM_INIT);
22    ap_uint<BIT_SUM_INIT> BOTTOM = Bounds_stream_fill.range(
23    BIT_RANDOM_NUMBER + BIT_SUM_INIT - 1, BIT_RANDOM_NUMBER);
24    ap_ufixed<BIT_RANDOM_NUMBER, 0> RAND;
25    RAND.range() = Bounds_stream_fill.range(BIT_RANDOM_NUMBER -
     1, 0);
26
27    for (u13_Data_T index = BOTTOM; index <= TOP; index++) {
28 #pragma HLS PIPELINE II=1
29 #pragma HLS LOOP_TRIPCOUNT min=1 max=6300 avg=16
30
31      Norm_weights_Data_T temp_value = acc_norm_weights_[index
     ];
32      if (temp_value >= RAND) {
33        saver_counter++;
34        sample_indx_stream.write(index);
35        break;
36      }
37    }
38  }
39  for (; saver_counter < NUM_PARTICLES; saver_counter++) {
40    sample_indx_stream.write(0);
41  }
42
43 }
```

Listing 7.4: Sampling

5. **Sample generation**

   The module task is to fill the sample memory with random pose. The module receives from the sampling stage an absolute index, which will be used to pick

the heuristic informations and from three different channels the angles. The code used to synthesise the hardware is the following :

```
#include "GenSample.hpp"
void Gen_Sample(stream<u13_Data_T> & sample_indx_stream,
    stream<angle_Data_T> & x_angle_stream,
    stream<angle_Data_T> & y_angle_stream,
    stream<angle_Data_T> & z_angle_stream,
    Mem_SAMPLE Samples[NUM_PARTICLES],
    Sample_abs_indx_Data_T Sample_abs_indx[NUM_PARTICLES],
    Sample_abs_indx_Data_T Sample_abs_indx_CPY[NUM_PARTICLES],
    BBOX all_box[6300], Mem_Heuristic Heuristics[6300],
    stream<randZ01> & random_z01) {

  // NOW I LOAD THE NEW ADDRESS
  for (u10_Data_T y = 0; y < Num_Particles; y++) {
    u13_Data_T sample_indx_fill = sample_indx_stream.read();
    ap_ufixed<18, 0> rand01 = random_z01.read();
    Sample_abs_indx[y] = sample_indx_fill;
    Sample_abs_indx_CPY[y] = sample_indx_fill;
    Mem_Heuristic curr_Heuristics = Heuristics[sample_indx_fill
   ];

    Samples[y].x_S = (ap_fixed<18 + BIT_POSE_INT, BIT_POSE_INT
   + 1>(
        curr_Heuristics.x_min + curr_Heuristics.x_max) >> 1);
    Samples[y].y_S = (ap_fixed<18 + BIT_POSE_INT, BIT_POSE_INT
   + 1>(
        curr_Heuristics.y_min + curr_Heuristics.y_max) >> 1);
    Samples[y].z_S = (ap_fixed<18 + BIT_POSE_INT, BIT_POSE_INT
   + 1>(
        curr_Heuristics.z_min
            + (curr_Heuristics.z_max - curr_Heuristics.z_min)
                * rand01));
    Samples[y].xroll_S = x_angle_stream.read();
    Samples[y].ypitch_S = y_angle_stream.read();
    Samples[y].zyaw_S = z_angle_stream.read();
  }
}
```

Listing 7.5: Sample Memory fill

The body is fixed in the centre of the bounding box, while the depth is selected randomly in the heuristic interval. The angles are generated with three different LFSR, which output value bounded to ± 0.5 and multiplied by π. The code to synthesise a generic angle is the following :

```
void Angle_Genx(stream<angle_Data_T> & angle_stream, u15_Data_T
    seed) {
  static u15_Data_T lfsr = seed;
```

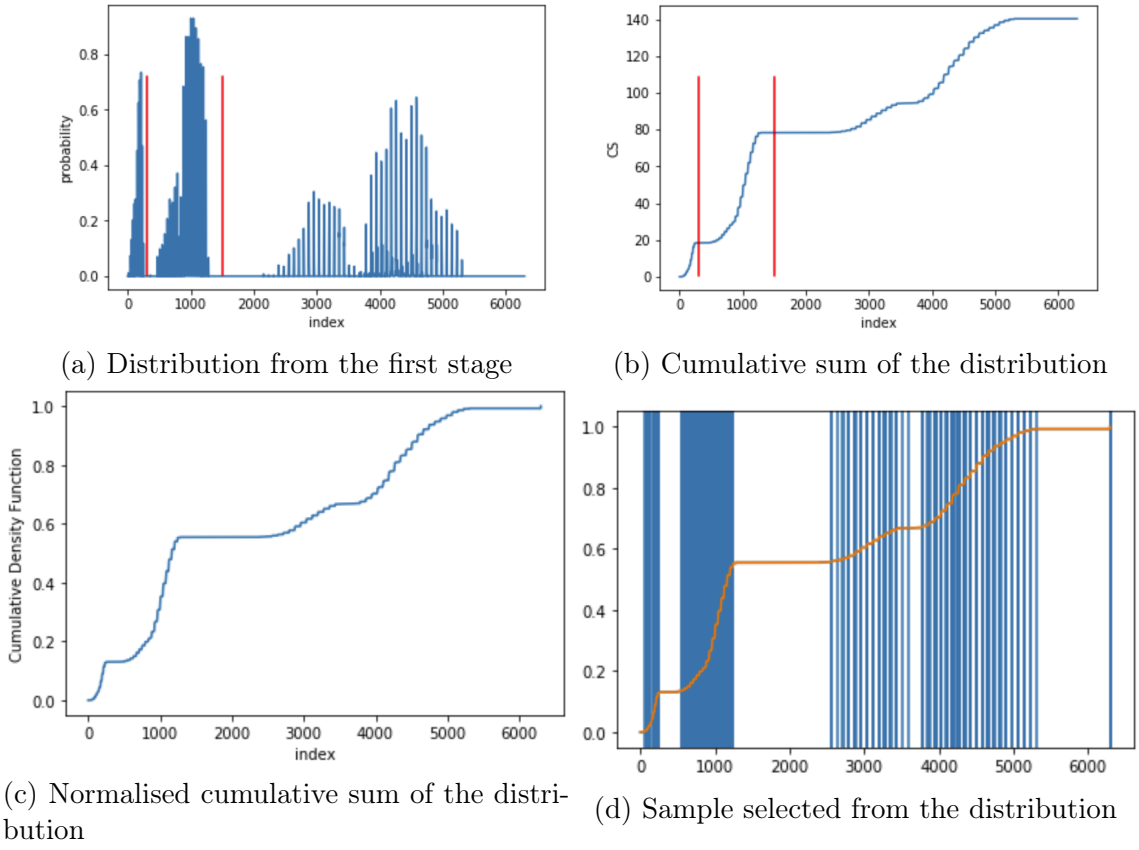
```
  for (u10_Data_T i = 0; i < Num_Particles; i++) {
#pragma HLS PIPELINE II=1
    u1_Data_T b_15 = lfsr.get_bit(15 - 15);
    u1_Data_T b_14 = lfsr.get_bit(15 - 14);
    u1_Data_T new_bit = b_15 ^ b_14;
    lfsr = lfsr >> 1;
    lfsr.set_bit(14, new_bit);
    ap_fixed<15, 1> rand_angle_temp;
    rand_angle_temp.range() = lfsr;
    angle_Data_T rand_angle_fill = (angle_Data_T) angle_Data_T(
        3.14159265359) * rand_angle_temp;

    angle_stream.write(rand_angle_fill);
  }
}
```

Listing 7.6: Angle generator

Results

The image collection number 7.10 shows the input data from the first stage and all the passages up to the sampling. Image 7.10a shows a real heatmap, flattened, which is converted to the image 7.10b and normalised to 7.10c. The image 7.10d shows which indices of the original distribution are selected during the initialisation.

The experimental result of the threshold analyser are shown in image 7.11c and 7.11d. The average of number of readings without the threshold tracker is 2520.27 versus 61.51 with it, leading to a speed up of 40.97 times. Note* The data shown are specific for the banana object inside the scene 1.6.



(a) Distribution from the first stage

(b) Cumulative sum of the distribution

(c) Normalised cumulative sum of the distribution

(d) Sample selected from the distribution

Figure 7.10: Data collected from the initialisation module

7.5 Render, Inlier and Re-sampling all together

The rendering, Inlier computation and Re-sampling functions are the core of the algorithm in terms of complexity, size and information elaborated. The transformation module generates the transformation matrices, for each sample, used by GPU module to correctly rasterize and shade object model. The image will be compared to the depth scene producing a output weight forwarded to the resampling stage. Since these three cores are strictly related a complex architecture was developed to reduce the data storage and speed up the overall operations, top RTL view can is shown in the image 7.12.

In figure 7.13 is shown the computational pipeline of the GPU and FPGA implementation for the weight calculation. The main difference between the two implementation is that each core inside instead of render and only after compare it to the image in the FPGA implementation this happen directly without storing the rendered image.

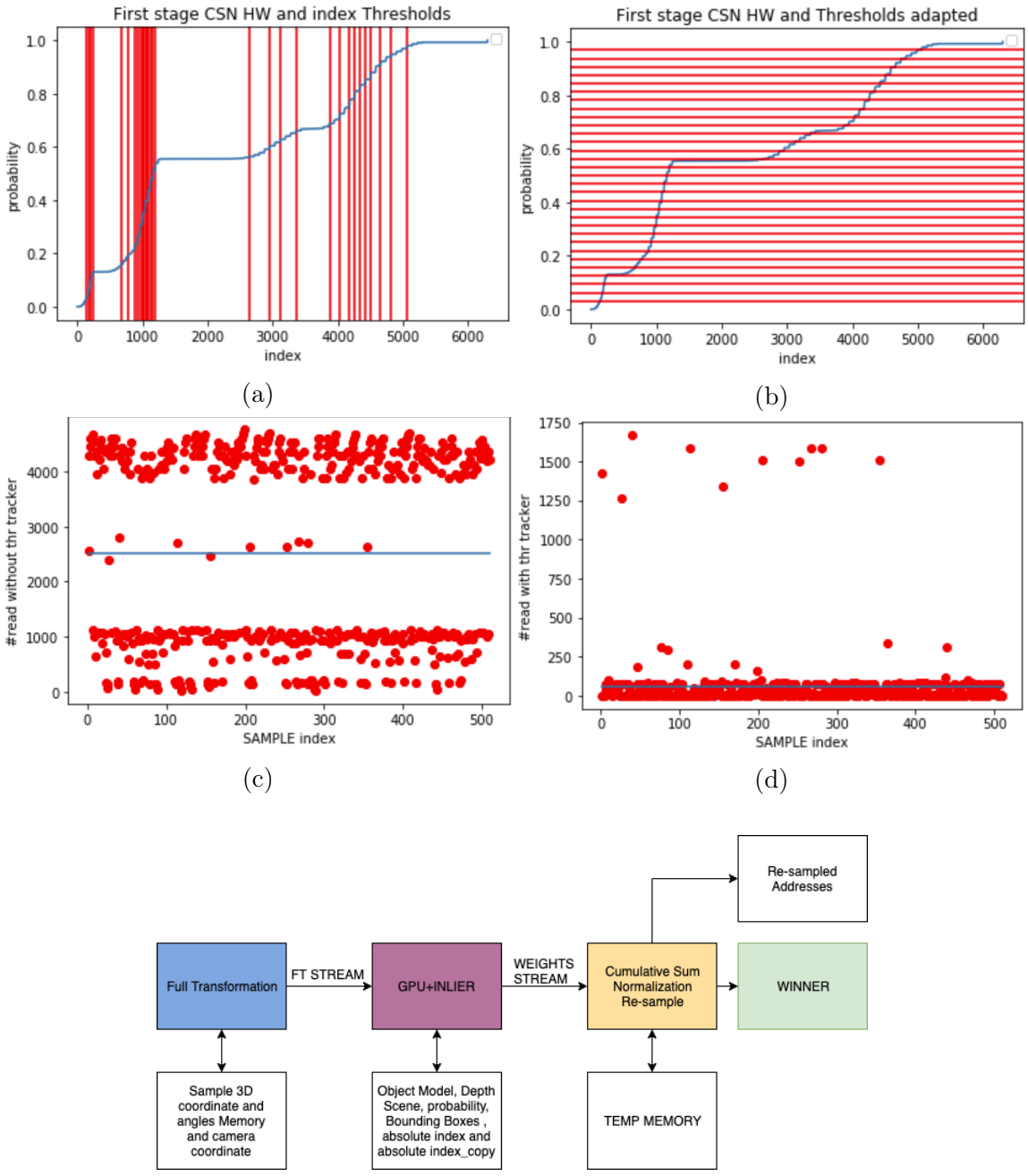


Figure 7.12: Top view of the particle filter.

7.5.1 Optimisations inside the architecture

In the preliminary version, the algorithm was working on one sample at time, storing the transformed object model in a temporary memory, rendering the sample in a

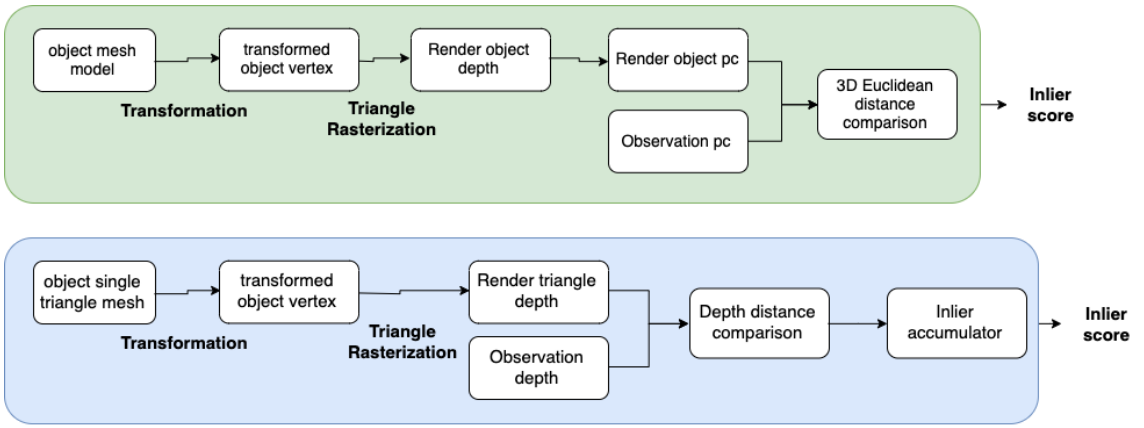


Figure 7.13: Particle filter dataflow difference between GPU and FPGA

memory of the same size as the original frame and to evaluate the inlier the entire depth image was scanned.

The new architecture as like the rest of the project was developed in terms of small computational modules. The main difference respect to the base version are the following differences :

- Fast, chained and low latency transformation matrix calculator
- No storing the transformed object model
- Reduced the size of the rasterisation space to the maximum bounding box size
- Sharing the same object model to more than one rasterization core.
- No Z-buffer
- Back face culling
- direct inlier evaluation

Soft inlier

The GPU original algorithm render an entire object, generates its point cloud and only after compare it to the point cloud of the scene with a 3D distance comparison. Each point difference lower than 5 mm is consider as an inlier and summed to the

previous values.

$$weight_i = \alpha \frac{Inlier_i}{N_{raster}} + \beta \frac{Inlier_i}{N_{depth}} + \gamma \rho_{FirstStage}$$

$$Inlier_i = \sum_{i=0}^{N_{raster}} \chi(||p_i - p_{scene_i}||) \tag{7.4}$$

$$\chi(x) = \begin{cases} 1 & x \leq 0.005 \\ 0 & x > 0.005 \end{cases}$$

As shown in 7.4 final weight is equal to the number of inlier over the valid render pixels plus inlier over the valid depth plus the probability of the first stage. All of these coefficients are multiplied by specific values in order to maximise the overall wight to one and redistribute the importance. In the hardware implementation the first step was to simplify the inlier evaluation by approximate the computation using the assumption that the (x, y) coordinate of the rasterized and scene point were the same if they belongs to the same image pixel.

$$||p_i - p_{scene_i}|| = \sqrt{(x_p - x_s)^2 + (y_p - y_s)^2 + (z_p - z_s)^2} \tag{7.5}$$

$$||p_i - p_{scene_i}|| \approx |z_i - z_{scene_i}|, \quad (x_p - x_s)^2 \approx 0 \quad and \quad (y_p - y_s)^2 \approx 0$$

With the approximation 2 subtractions, 2 additions , 3 multiplications and one square root are removed and the need of a point cloud for the object.

Furthermore, to extract more informations about the object the $\chi(|\Delta d_i|)$ function was transformed to a discontinuous function to a continuous one by selecting two distance thresholds and interpolated them. Since this computation should not increase both hardware and time only shifting and additions wants to be used. First step to design the soft-inlier calculation was to find two reasonable powers of two close to 5 mm.

$$Thr_1 = 0.00390625 = 2^{-8} \approx 4mm$$

$$Thr_2 = 0.0078125 = 2^{-7} \approx 8mm$$

To map this information from zero to one we need to find the slope that remap the input to that specific interval

$$\alpha = \frac{\chi(Thr_2) - \chi(Thr_1)}{Thr_2 - Thr_1} = \frac{0 - 1}{2^{-7}} = -128$$

which is actually 8 point shift to the depth difference value. the final $\chi(|\Delta d_i|)$ uses only one adder more but gives us a more precise value to describe the object.

$$\chi(x) = \begin{cases} 1 & x \leq Thr_1 \\ 1 + \alpha x & Thr_1 < x < Thr_2 \\ 0 & x \geq Thr_2 \end{cases}$$

Memory sharing and reduction Once the system was implemented to a multi-core architecture a problem was noticed. The depth fragment distribution was adding a huge delay was due to the huge amount of data transfer and to the low clock speed. However, some correlation between the inner cores/particles arise, like that all them needs the same :

- Depth image : Each core needs to compare the rendered object with the corresponding depth.
- Object model : Each core needs to render the body its mathematical description.

If a copy of all of these information is provided to each core is quite easy to running out of memory and consequently the number of core cannot increase. To have a flexible and scalable system the modules can share :

- Depth image : It is possible to reduce the depth memory inside the rendering module to the max bounding box possible and thanks to a specific engine the time to fill all of them can be drastically reduced. This external module reads, assemble all the depth fragment and distribute the depth pixel to corresponding rendering core, reading the depth memory only once.
- Object model : It is possible thanks to an external process that read the original object model, collect the triangle and send a copy to all of them.

Furthermore, an extra reduction in terms of memory consumption was made in the project. Thanks to the back face culling almost of the fifty percent of the triangles are removed, because intrinsically occluded. The proposed architecture exploits this propriety to calculate online the weight, without storing the rendered body, by directly compare the pixel value to the correspondent depth pixel. The approximation done with the back face culling is not perfect since it removes only the faces pointing in the same direction of the camera, and this can be a problem in the edge of the objects and in the one which has not fully convex shape.

This inconvenient alters the value of the weight by increasing or decreasing it bit, but it also reduces a lot the number of memory consumption for each module, since there is no need to store the rasterised object . Figure 7.14 shows the FPGA and GPU rendering dataflow.

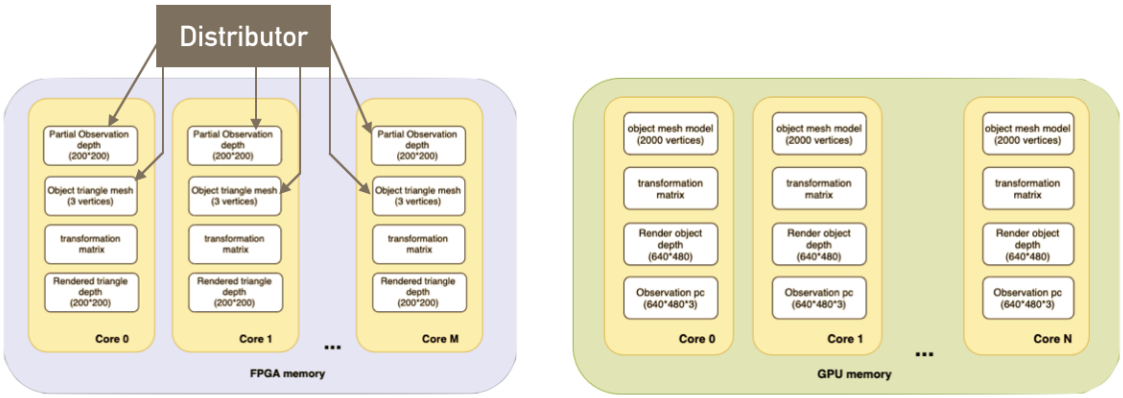


Figure 7.14: Rendering pipeline, FPGA and GPU differences

7.5.2 The full transformation matrix chain

The module has the task to produce the transformation matrix to be used inside the rendering core $FT = PCTrRzRyRx$, and to improve the latency and the hardware consumption this module chains all the matrix multiplication across the samples. As a first step is possible to notice that full expression of the final matrix is $FT = PCTrRzRyRx$ can be grouped in partial multiplications $FT = (PC) * (TrRz) * (RyRx)$ and each of that multiplications can be optimised independently :

- The camera times the projection matrix is not worth to optimise it because the camera one is mostly different from zero and it have to be calculate only once whereas it is common to all the samples.

- The translation with the rotation on the z axis can be joined without the need for multiplication, as shown in equation 7.6.

$$\underline{\mathbf{TrRz}} = \begin{bmatrix} cos(\beta_z) & -sin(\beta_z) & 0 & x_0 \\ sin(\beta_z) & cos(\beta_z) & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.6}$$

- The rotation matrix around y and x can be merged just with four multiplication, as shown in equation 7.7.

$$\underline{\mathbf{RyRx}} = \begin{bmatrix} cos(\beta_y) & sin(\beta_y)sin(\beta_x) & sin(\beta_y)cos(\beta_x) & 0 \\ 0 & cos(\beta_x) & -sin(\beta_x) & 0 \\ -sin(\beta_y) & cos(\beta_y)sin(\beta_x) & cos(\beta_y)cos(\beta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.7}$$

Applying this consideration the overall number of multiplication is reduced to zero for the $TrRz$ matrix and to four for the $RyRx$ one. The architecture developed was the one the transformation matrix generator, figures 7.16 and 7.15 show the RTL view and the timing diagram. The module itself has only one output, there will be a next module with the task to redirect the data flow.

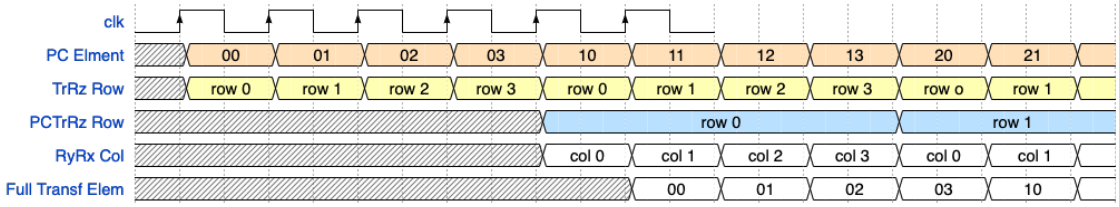


Figure 7.15: Timing for the full transformation matrix

The architecture in the image 7.16 was designed to archive high throughput. The only input is the BRAM containing all of the information about the current samples and the output is the matrix element provided one by one. The matrix pipeline can be stalled and reactivated only when need, in this way there is no reason to store all the transformation matrix, just the one currently used by the rendering cores. The improvements is possible since the $TrRz$ and $RyRx$ can be produced without any matrix multiplication and independently evaluated. The output element is constructed by multiply an element of the PC times the $TrRz$, row by row. The output is in the form of a row which multiply by the column of $RyRx$ results in the final transformation matrix. The has the advantage that the multiplication between the two trigonometric function can be hidden during the transmission of the first column and the reading of the second angle.

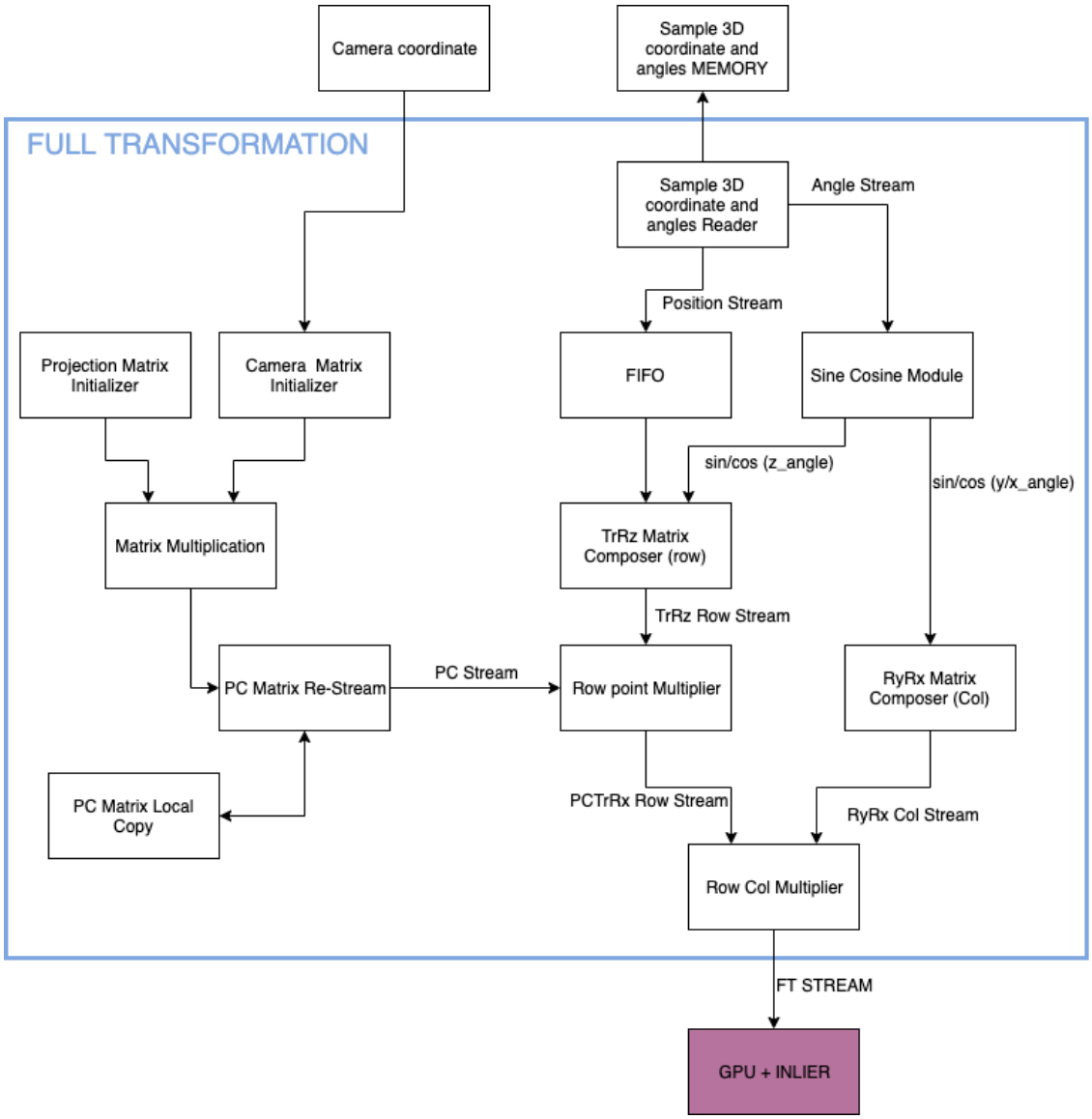


Figure 7.16: RTL view for the full transformation matrix pipeline

7.5.3 GPU and INLIER

The architecture proposed, shown in figure 7.17, is composed by the following modules:

- **Vertex Multiplier** : It load the points, assemble a face for the object model and transmit a copy of it to all the rendering cores.
- **Distribute FT** : This module reads the output of the full transformation

matrix module an redirect the element to the correspondent rendering core.

- **Depth fragment and Sample Info Distribute** : It collect all the bounding box informations, create a map of the depth pixel that have to be transmitted and send them with the equivalent bounding box information.
- **Raster Core (partial inlier)** : It transforms and rasterize the triangle directly on the depth portion outputting the numerator and denominator of the final weight.
- **Weigh/Inlier merger and final composition**: This module reads all the outputs from all the raster cores, generates the final weight, concatenate it to the corresponding relative index and send this information to the sorting network. Furthermore, the module was implemented to use reduce the number of division module because otherwise each rasterization core needs one.

Depth Distribution

As previously described, now that the system is multi-core, the depth information must be redistributed in an efficient way because inside each rasterization core it is not stored the entire image but only a fragment of it and the time for the distribution of the information is directly proportional to the number of cores. For example in this project case with five rasterization cores ,a the clock frequency of $200MHz$ and a depth fragment of $200x200$ the overall time will be $1ms$, if distributed with a sequential approach, per sample in raster core. Moreover, is important to point out that this is the time only for the writing and do not consider the reading time, $1ms$, leading to an overall delay for all the sample :

$$Time_{iteration} = D_H * D_W * (\#Clk_{read} + \#Clk_{write}) * T_{clk} \frac{N_{sample}}{M_{core}} = \\ 200 * 200 * 2 * 5 \frac{510}{5} ns = 204ms \tag{7.8}$$

To reduce the number of readings, memory access delay, and the overall latency an algorithm was designed to join the information from each raster core, as shown in figure 7.18.

The implemented algorithm was inspired by the assumption that more channels are present in the design more likely is that exist a depth fragment portion shares. The assumption at the beginning was supported also from the neural network predictions, almost right, and from the certain certainty that going on with the iterations the bounding boxes converge to the same depth area. All of these assumptions allows to create an algorithm which read the depth image only one time for many particles reducing both the time and power consumption.

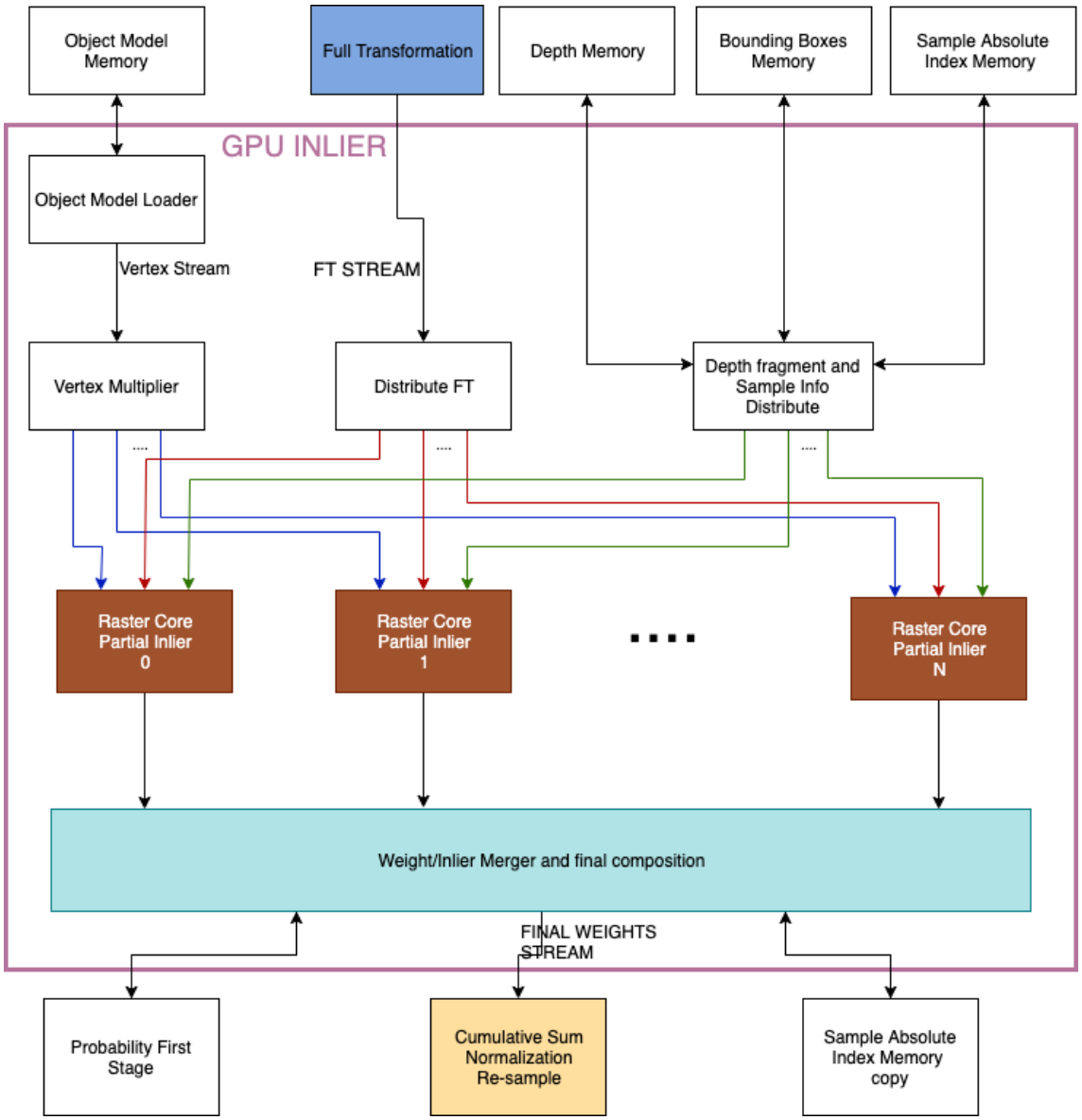


Figure 7.17: GPU and INLIER architecture

To archive the proposed algorithm the bounding box centre was stored, saturated to the image limits, sorted by the row coordinate and merged by the column one. Since the centre is related to the bounding box size could be different respect to the one of the depth fragment. The saturation transform the middle point if its distance respect to the edge of the image is lower than the depth fragment half. The following pseudo code is a small description of the actual algorithm :

```
1 for (u7_Data_T x = 0; x < Num_sample_x_raster_core; x++) {
```

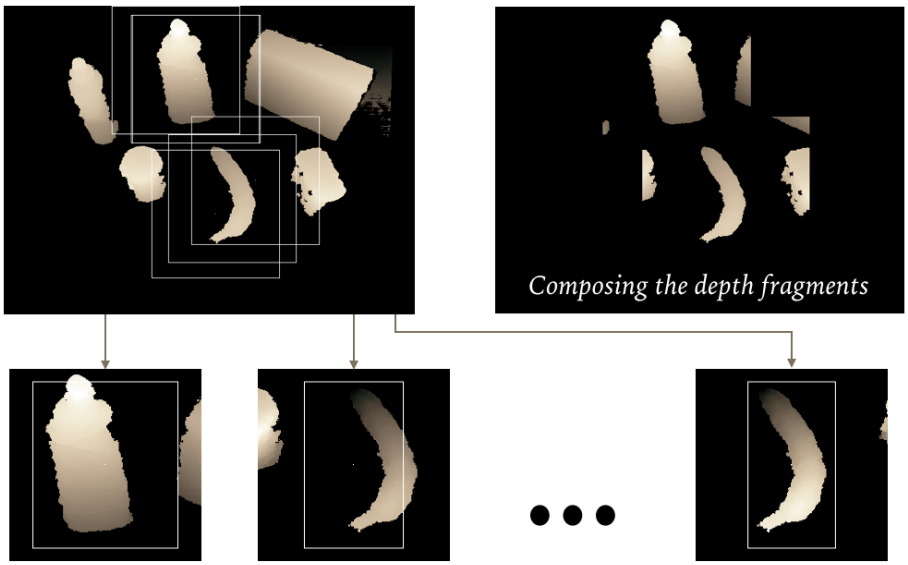


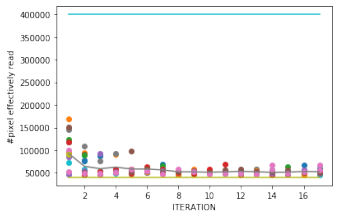
Figure 7.18: Depth distribution with information sharing.

```
2     for(u4_Data_T indice = 0; indice < Num_Core_Raster; indice++) {
3     Centre = read(Centre_BB);
4     Saturate(Centre,WIDTH, HEIGHT);
5     //this insertion sort limits its sortind dimension according to
      the current size
6     Limited_Insertion_Sort(Centre_rowSort, Centre, indice);
7     }
8     Create_Rowinterval(Centre_rowSort,row_interval,
    row_interval_size);
9     //FIND WHICH CHANNEL BELONGS TO A SPECIFIC INTERVAL
10    Find_channel(Centre_rowSort,row_interval_list,
    row_interval_list_size,Ch_list);
11    Merge_Xinterval(col_interval_size, col_interval_list, Ch_list,
    row_interval_list,
12           row_interval_list_size, Centre_rowSort)
13
14    for (int index = 0; index < row_interval_list_size; index++) {
15      for (int row = row_interval_list[index];
16           row < row_interval_list[index + 1]; row++) {
17        for (int interval_index = 0;
18             interval_index < col_interval_size[index];
19             interval_index++) {
20           for (int col = col_interval_list[index][interval_index].
    min;
21                col < col_interval_list[index][interval_index].max;
22                col++) {
23             // load the temporary depth
24             Depth_Data_T temp_depth = Depth_scene[row][col];
```

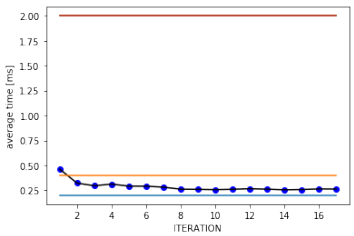
```
25              for (int ch_counter = 0; ch_counter < Num_Core_Raster;
26                   ch_counter++) {
27                // CHECK FOR ALL THE CHANNELS
28                if (row >= Centre[ch_counter].y_begin
29                    && row < Centre[ch_counter].y_final
30                    && col >= Centre[ch_counter].x_begin
31                    && col < Centre[ch_counter].x_final) {
32                  Depth_stream[ch_counter].write(temp_depth);
33                }
34              }
35            }
36          }
37        }
38      }
39  }
```

Listing 7.7: Depth distribution pseudo code

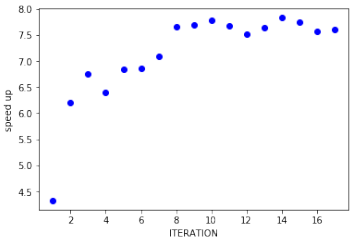
Results The data collected, figure 7.19a and 7.19b, show that the assumption made at the design stage was right and figure 7.19c states the overall speed up obtained.



(a) # of pixel read from the Depth BRAM



(b) Time to fill all the raster cores



(c) Overall speed up respect the version without the processing

core	Run Time	
1	204	ms
5	85	ms
10	55	ms

Table 7.2: Run time collected from the board

Note* the results provide are evaluated with ten rasterization cores.

Rasterization

The architecture, shown in figure 7.20, can be analysed in four main modules :

- Triangle transformation : This module load the transformation matrix and transform all triangle of an object model.
- Depth recovery : This module recover the exact depth from the distortion introduced by the projection matrix
- Depth comparison : This module compare the input pixel with the corresponding depth one. This module outputs the numerator and denominator needed for the weight calculator.

The rasterization module implements also the back-face culling,which performances are shown in image 7.21. The Overall Faces over Rendered ones is equal to 2.06 .

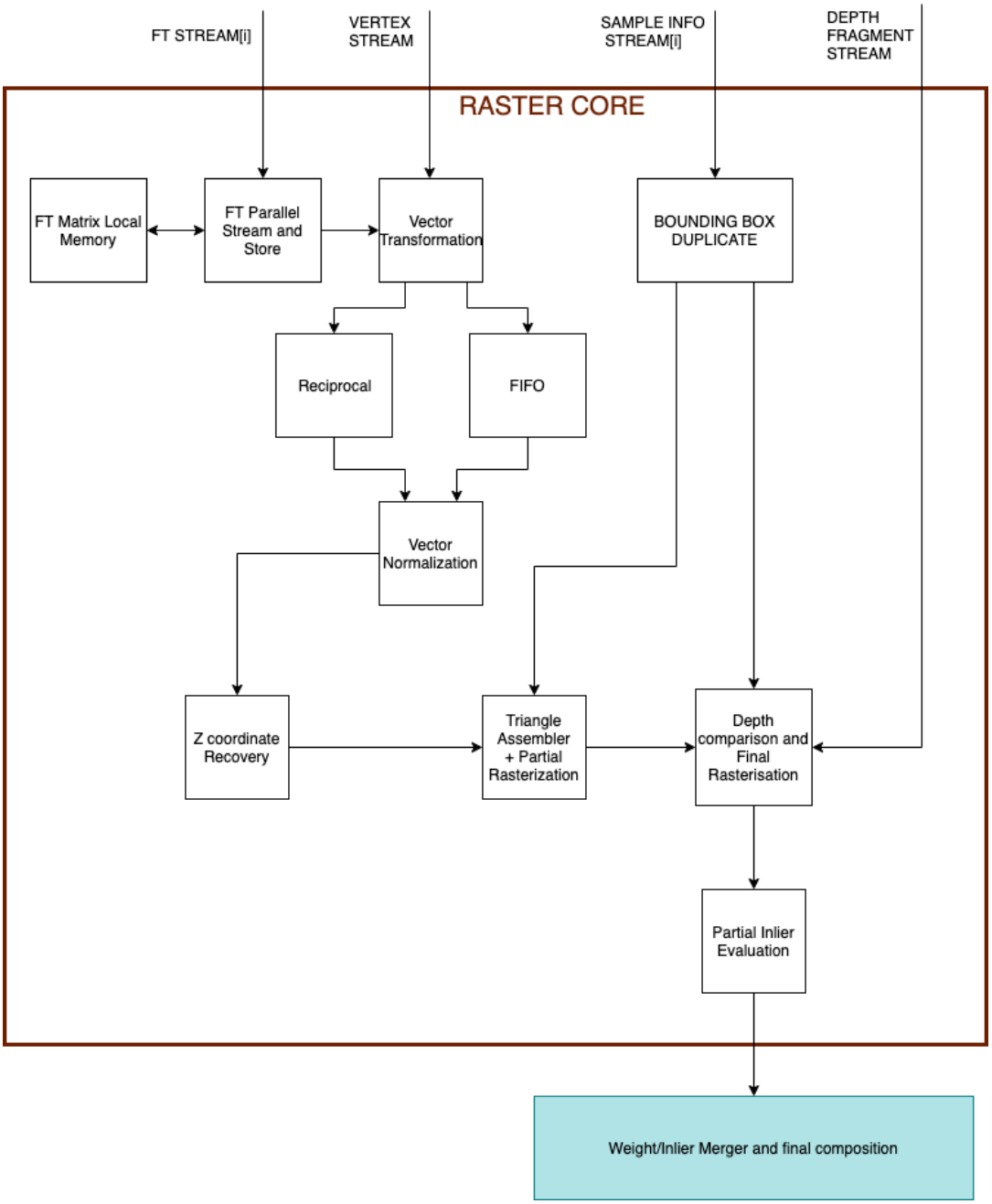


Figure 7.20: Rasterization architecture

Weight merger

The module is just a division core that evaluates the final weight for each particle and outputs to the resampling stage the value with the relative index concatenated.

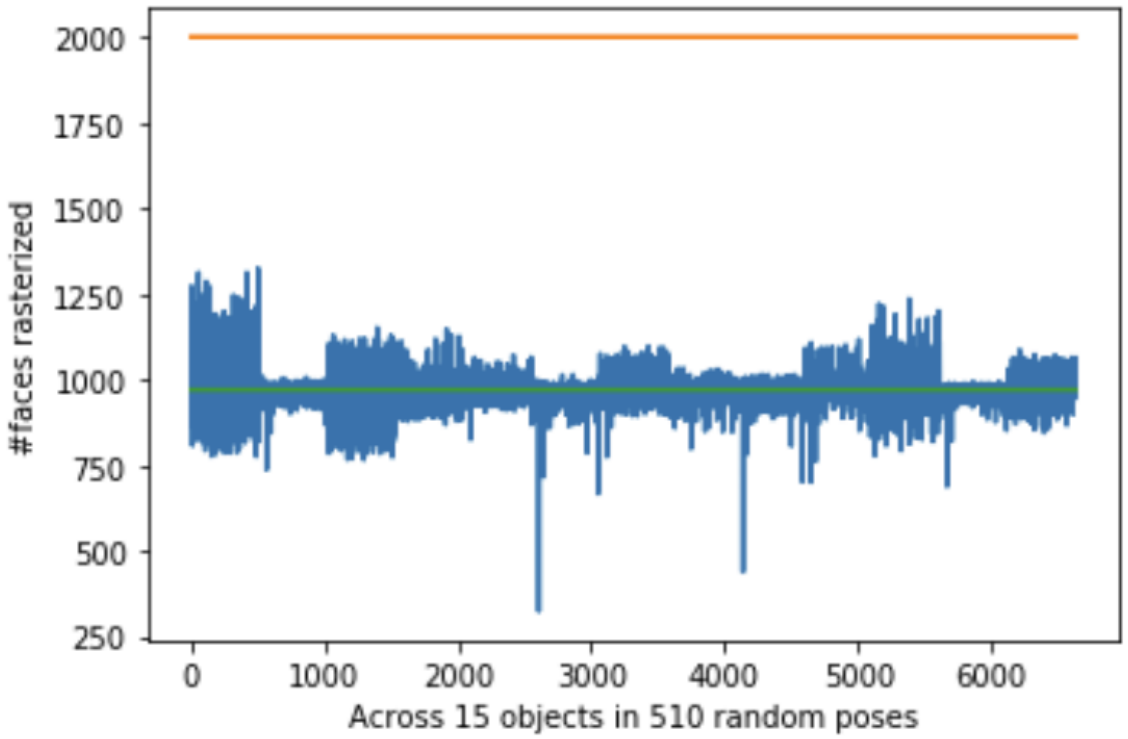


Figure 7.21

```
void weight_merger(stream<Raster_Core_Output> (&Serial_weights)
    [10],
     Sample_abs_indx_Data_T Sample_abs_indx_CPY[NUM_PARTICLES],
     FS_W_fix_Data_T all_prob[6300], stream<Inlier_Output> & Inl_Out
    ) {
   for (int x = 0; x < Num_sample_x_raster_core; x++) {
     for (int core = 0; core < Num_Core_Raster; core++) {
#pragma HLS PIPELINE
       weight_address_Data_T address = core + x * Num_Core_Raster;
       Sample_abs_indx_Data_T curr_abs_index = Sample_abs_indx_CPY[
    address];
       Raster_Core_Output Serial_weights_fill =
           Serial_weights[core].read();
       FS_W_fix_Data_T curr_probability = all_prob[curr_abs_index];
       Inlier_Output Outfill;
       Outfill.weight =
           Numerator_Data_T(
               ((Serial_weights_fill.Numerator)
                   / Serial_weights_fill.Denominator
                   + curr_probability) >> 2);
       Outfill.address = address;
       Inl_Out.write(Outfill);
```

```
20      }
21    }
22 }
```

Listing 7.8: Weight merger

7.5.4 Re-sampling

The architecture developed, shown in figure 7.22 to sample the particles is quite similar to the one developed for the object pose initialisation. The information arrives sequentially, they are accumulated and sorted through a sequential sorting network. Only after all the samples are arrived the normalisation coefficient is calculated and the normalisation process started. The final distribution pass through a threshold tracker and stored in a temporary variable, the one used in the initialisation core. The module will pick some samples and write their relative index to a memory, which will be used later to generate the diffused poses.

Sorting network The sorting module in the original version took a lot of time and data movement. The architecture proposed in figure 7.23b is able to sort online the weights and produce an array of relative memory indices in the diffusion stage. The sorting network is subdivided in smaller modules called macro sorting cells, which is feed with the information coming from the inlier function, which outputs the weights in a sequential manner . The macro sorting cell is a shift register with conditional insertion, as shown in 7.23a, designed to contain an array always sorted. The basic idea is that every time an element go inside the macro sorting cell another value must go out and pass to the next cell or to the final output. This algorithm is very similar to an insertion sort, and the division in macro cells allows us to customise the number of elements to be effectively stored.

The approach with the macro cell allows to modify the number of particles to be sorted, because in each cell is possible to store M weights. According to the ratio of particles that wants to be used during the re-sample stage is possible to increase or decrease the number of cells needed. In this way is possible to archive a sorting of N elements concurrently to the inlier evaluation.

Re-sampling and threshold tracker

The type of resampling used is an importance sampling, which means to generate a random number from a uniform distribution and pick the first element that overcome that value. As was happening in the initialisation also here to find the value the array must be scanned from the zero address. Also in the resampling architecture is possible to add the threshold tracker module to reduce the number of readings, figure 7.24a and 7.24b.

From the data shown in 7.25b and 7.25a, the average of number of readings without the threshold tracker is 375 versus 8.34 ,with it. The overall speed up is 44.93 times.

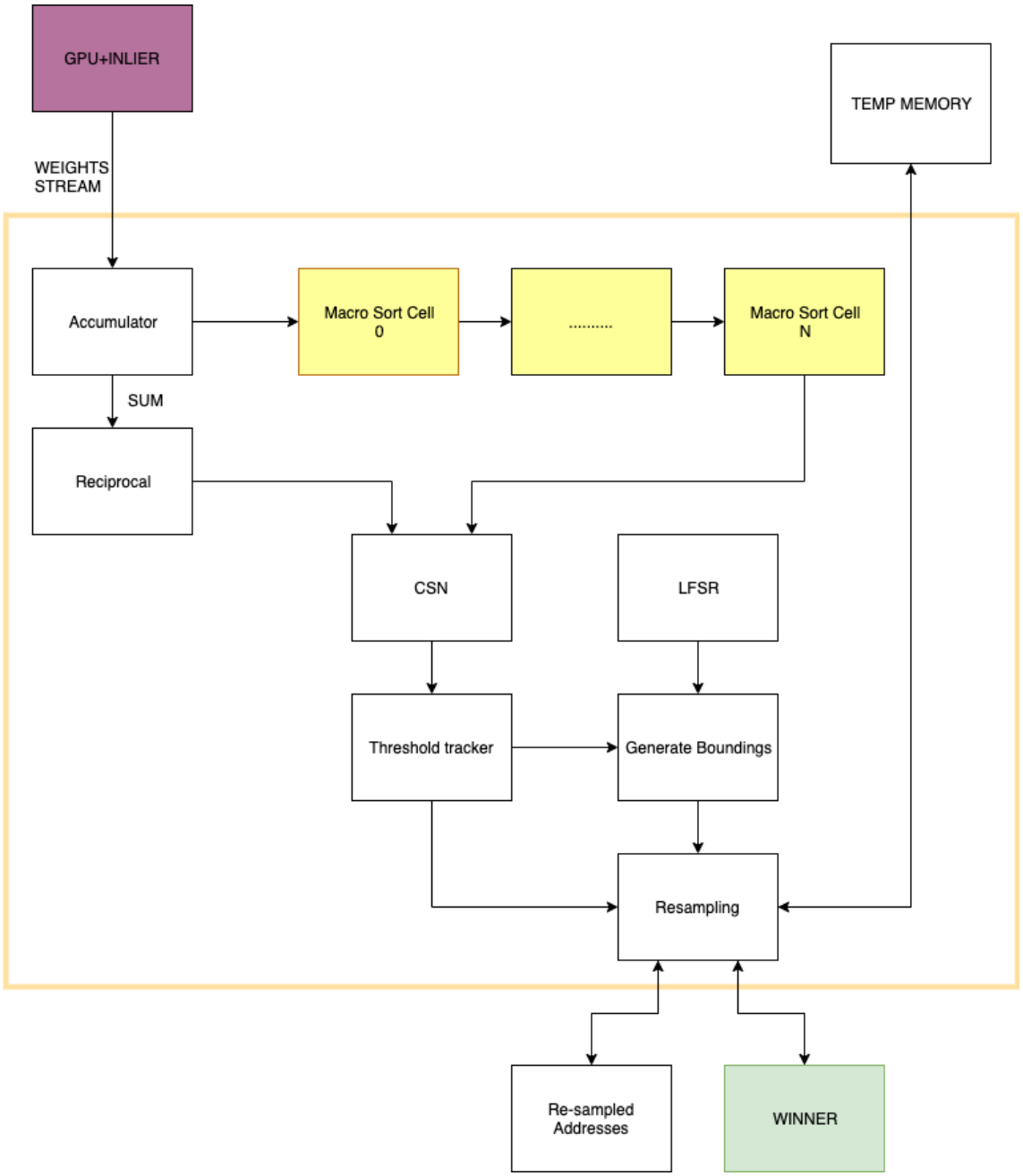
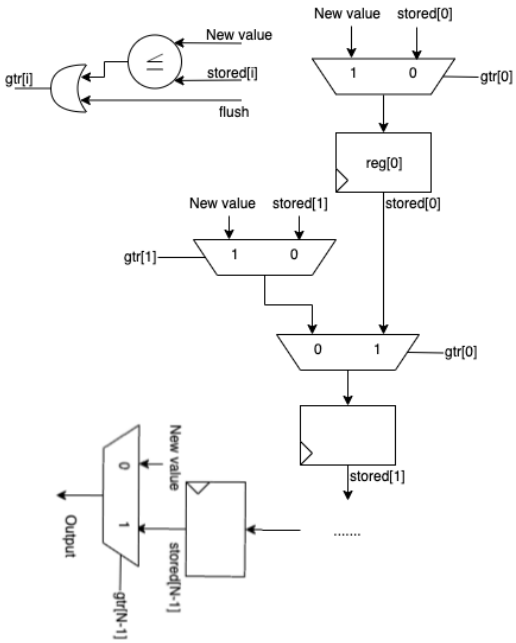


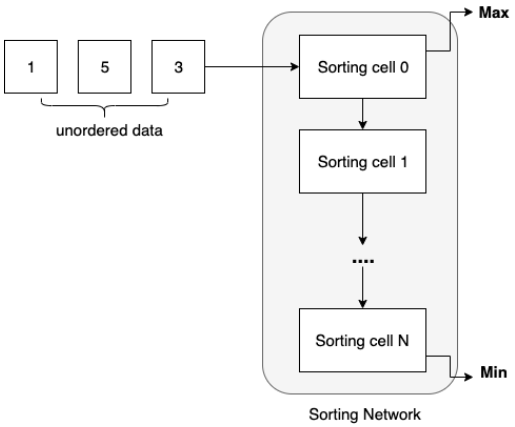
Figure 7.22: Re-sampling architecture

7.6 Diffusion

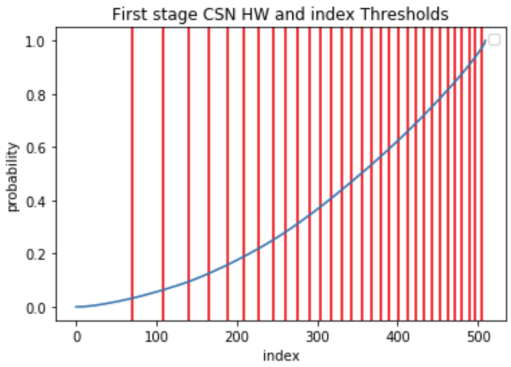
The diffusion stage has the task to modifies the re-sampled elements to produce a new batch with similar position of the most likely. In order to avoid problem of



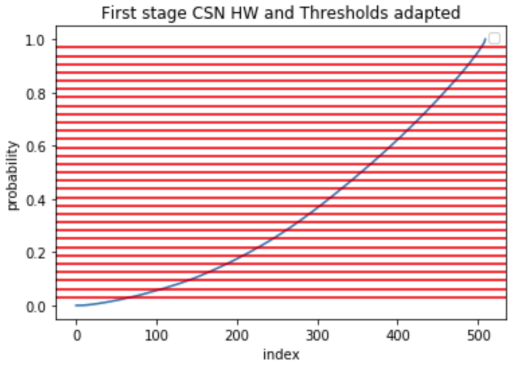
(a) Sorting network, general datapath



(b) RTL view of the serial sorting network



(a) Threshold indices positions respect to the CSN

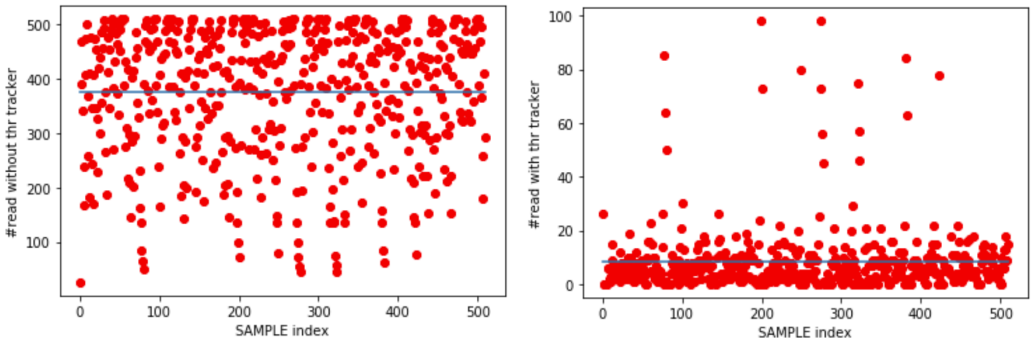


(b) Threshold value positions respect to the CSN

overwrite, since the original original array is not sorted, the particle filter works with two sample memories, which alternate during the iteration as start and ending point.

Figure 7.26 shows the RTL view of the diffusion module. From a set of LFSR uniform random numbers are produced, these can be divided in two main channels. The first group goes to the diffuse type generator, which select a type of diffusion between :

- Translation for x,y,z or all of them.



(a) Number of readings needed without the Threshold tracker

(b) Number of readings needed with the Threshold tracker

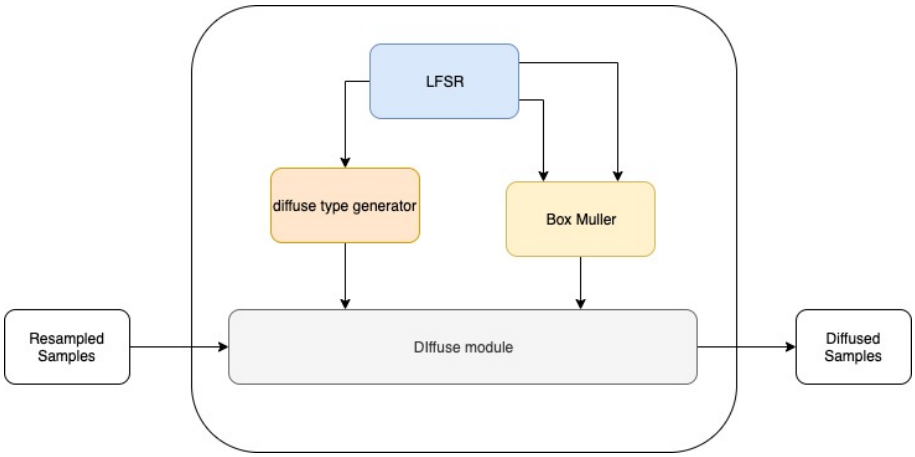


Figure 7.26: Diffusion architecture.

- Rotation for rho,pitch,yaw or all of them
- Add 180 degree to one of rho,pitch and yam
- Moving the bounding box across and among the original heatmap

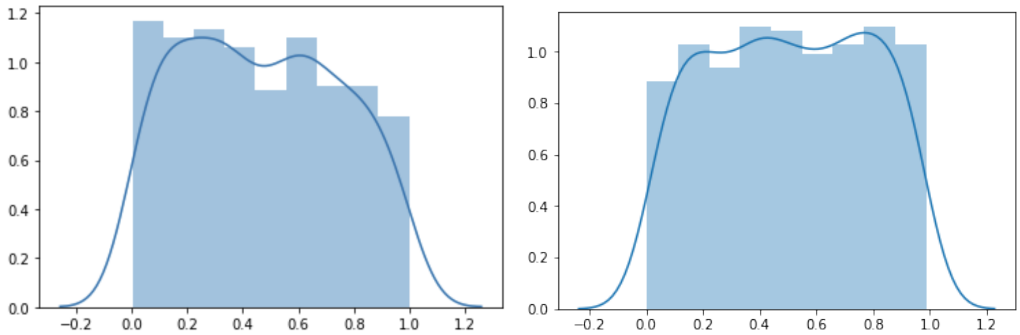
The second group goes to the Box-Muller algorithm, which generate a couple of number distributed in normal distribution starting from two uniform ones. The box Muller works independently from the diffusion stage and is able to produce a couple of Gaussian numbers each clock cycle. The output noise is modulated always inside the same core to change the final variance. The distortion's effect decrease linearly if the maximum weight increase.

$$translation_{var} = (1 - winner.weight)var_0$$
$$rotation_{var} = (1 - winner.weight)var_1$$

Furthermore, respect to the original algorithm this decay was changed to a linear decay, easier than an exponential one, directly proportional to the maximum probability of the sample.

Results

In this small paragraph some distribution of random number will be proposed to validate the design. Figure 7.27a and 7.27b are two histograms of the random number used in one iteration of the particle filter. Since the number used is not infinite is possible to notice a distortion in the envelope.



(a) Uniform distribution from initialisation

(b) Uniform distribution from resampling

Figure 7.28a shows a the histogram of one variable generated by the box Muller transformation. The characteristic of the distribution are :

- Mean: 0.006161
- STD: 0.958241
- Skewness: 0.049544 (asymmetry parameter, unbalanced distribution)
- Kurtosis: 0.139801 (how much of the distribution is in the tail)

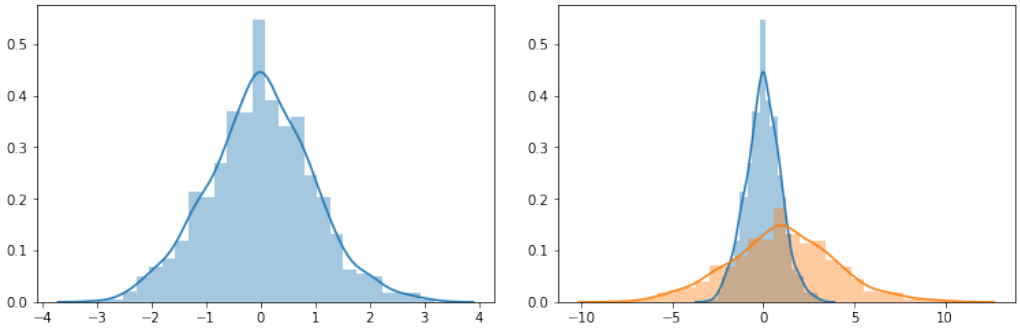
To test if the null hypothesis cannot be rejected two test were executed :

1. D'Agostino K2 Test : uses the Skewness and Kurtosis to understand if it is a Gaussian distribution. This tests generates stat = 1.35169 , p-value = 0.508726. The understand if the samples looks Gaussian distributed the p-value should be greater than alpha, 1e-3.

2. Shapiro-Wilk Test : is characterised by the statistics : 0.997846 and p-value : 0.224071, with a chosen alpha of 0.05.

With both tests the samples look Gaussian. Figure 7.28b shows the same distribution modified by mean and variance.

- Blue : Mean: 0.006161 and STD: 0.958241
- Mean: 1.018485 and STD: 2.874723



(a) (b)

7.7 Hardware and power consumption

In this section are shown two example of design. Respectively with five and ten rasterization core. Legend of image 7.31:

- BLUE: 5 GPU and INLIER CORE
- PINK: SORTING NETWORK
- YELLOW: INIT
- DARK PINK: POST-PROCESSING
- ORANGE: AXI
- DARK GREEN: FT CREATE
- LIGHT GREEN: DIFFUSION

Legend of image 7.34

- PINK: 10 GPU + INLIER CORE
- BLUE: SORTING NETWORK

5 Cores

Resource	Utilization	Available	Utilization %
LUT	111433	274080	40.66
LUTRAM	1784	144000	1.24
FF	127899	548160	23.33
BRAM	411.50	912	45.12
DSP	274	2520	10.87
BUFG	8	404	1.98

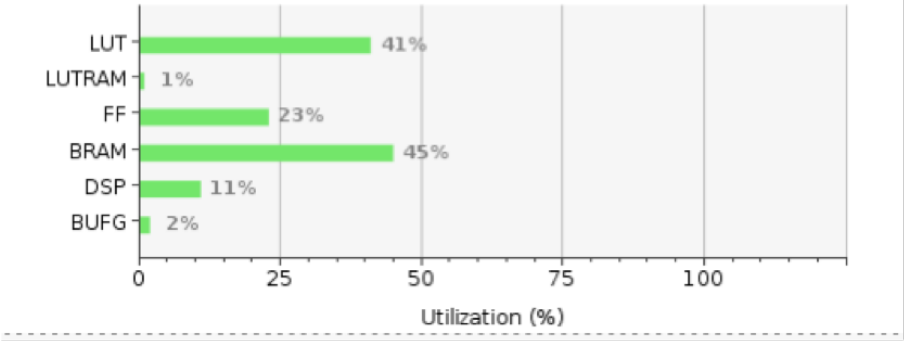


Figure 7.29

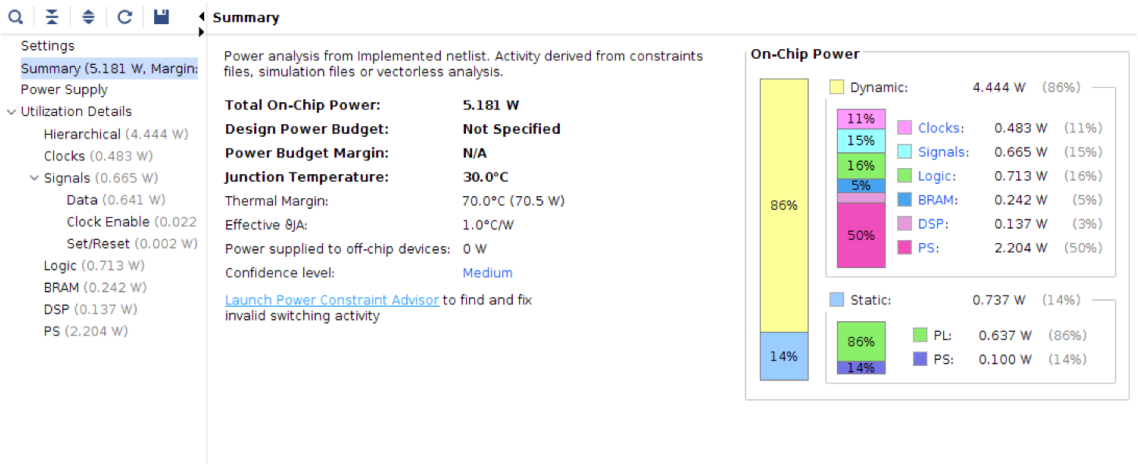


Figure 7.30

10 Cores

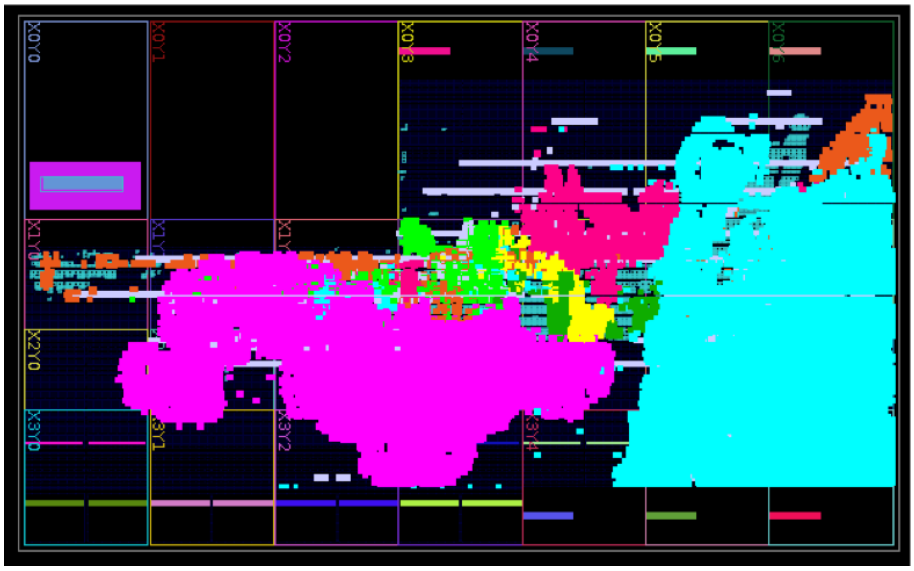


Figure 7.31

Summary

Resource	Utilization	Available	Utilization %
LUT	165628	274080	60.43
LUTRAM	2872	144000	1.99
FF	190560	548160	34.76
BRAM	605	912	66.34
DSP	472	2520	18.73
BUFG	16	404	3.96

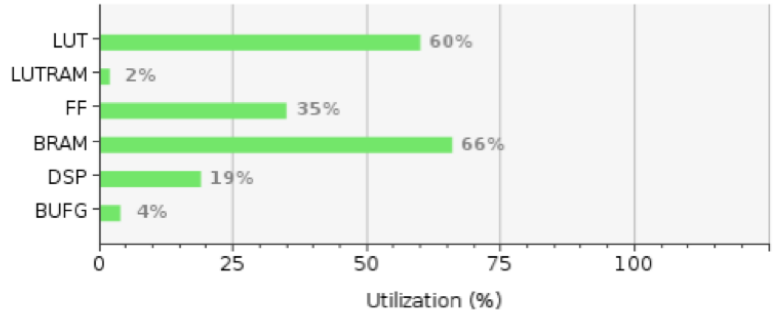


Figure 7.32

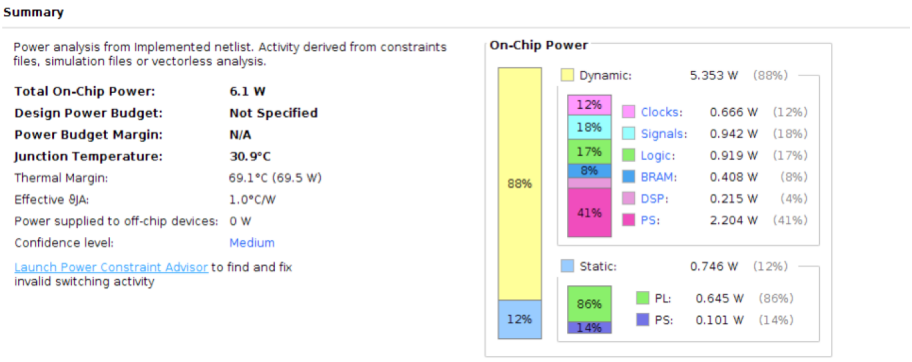


Figure 7.33

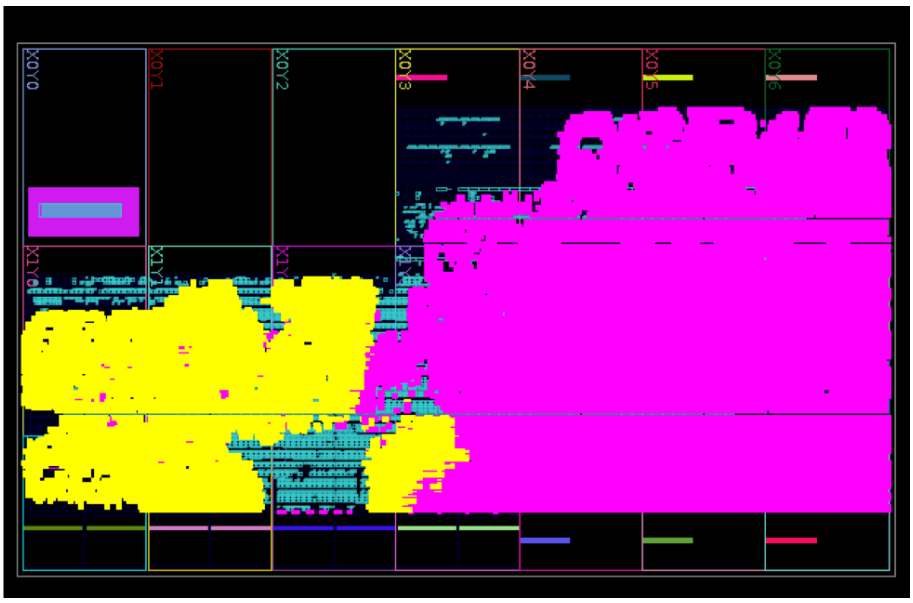


Figure 7.34

7.8 Conclusions

This thesis aimed to create from the high level algorithm a complete architecture for FPGA. Based on the final design is possible to state that develop individual cores from an RTL prospective was quite useful. The flexibility also from the architectural point of view was reached since the number of inlier core can be changed according to the FPGA size. Furthermore, thanks to the low power consumption the execution of this specific particle filter inside any embedded platform. **Time performances**

The GPU used to compare is a NVIDIA Titan Xp, high power GPU not designed for embedded systems. By comparing the performances, shown in 7.3, is possible

core	Frequency[MHz]	1^{st} iteration[ms]	50^{th} iteration[s]	Average time[ms]
5 (FPGA)	**200**	177	6.31	**120**
10 (FPGA)	**200**	115	4	**80**
625 (GPU)	**1770**	30	1.5	**30**

Table 7.3: Time performance FPGA and GPU

to notice that even if the FPGA uses only 10 cores, respect to 625 of the GPU, the final slow down is only 2.66. By looking at image 7.35 is also possible to notice that the execution time, in the FPGA, decrease as the iteration increase, while the GPU one is always constant.

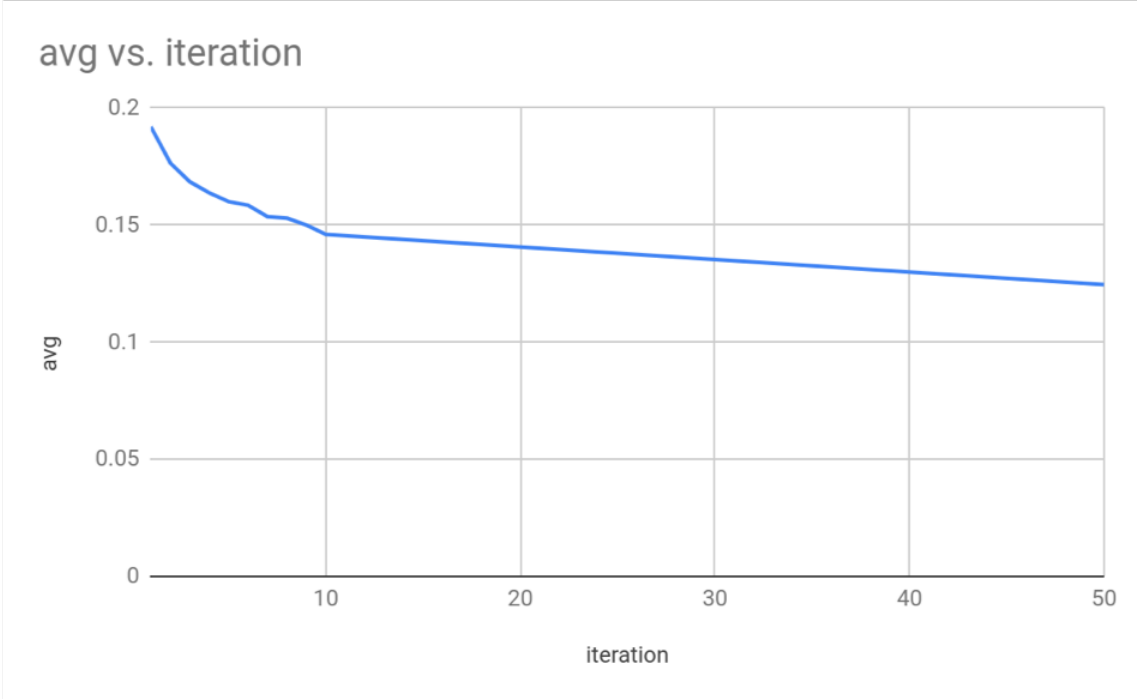


Figure 7.35: Iteration time vs number of iteration for five cores

Bibliography

[1] Xiaotong Chen, Rui Chen, Zhiqiang Sui, Zhefan Ye, Yanqi Liu, R. Iris Bahar, Odest Chadwicke Jenkins , *GRIP: Generative Robust Inference and Perception for Semantic Robot Manipulation in Adversarial Environments*. https://arxiv.org/abs/1903.08352.

[2] Narrow AI : https://en.wikipedia.org/wiki/Weak_AI

[3] Rajeev Jayaraman, Xilinx Inc, 2001 https://www.doc.ic.ac.uk/ wl/teachlocal/arch/killasic.pdf

[4] Xilinx Vivado Design Suite User Guide : High-Level Synthesis [UG902 (v2019.1) July 12, 2019]

[5] What data scientist should know about deep learning (Andrew Ng): https://www.slideshare.net/ExtractConf/andrew-ng-chief-scientist-at-baidu

[6] Universal approximation theorem : https://en.wikipedia.org/wiki/Universal_approximation_theorem

[7] Bayes' theorem : https://en.wikipedia.org/wiki/Bayes%27_theorem

[8] Convolutional Neural Network : https://en.wikipedia.org/wiki/Convolutional_neural_network

[9] The top-5 error rate in the ImageNet : https://devblogs.nvidia.com/mocha-jl-deep-learning-julia/image1/

[10] Particle filter : https://en.wikipedia.org/wiki/Particle_filter

[11] Importance sampling : https://en.wikipedia.org/wiki/Importance_sampling

[12] Tomas Akenine-Moller : Appendix A. Fixed-point Mathematics http://fileadmin.cs.lth.se/cs/Education/EDA075/notes/mgh_appA_fixed.pdf

[13] 7 Series FPGAs , Memory Resources : https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[14] UltraScale Architecture, DSP Slice : https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[15] Introduction to FPGA Design with Vivado High-Level Synthesis: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[16] CORDIC : https://en.wikipedia.org/wiki/CORDIC

[17] D'Agostino, R. and Pearson, E. S. (1973), "Tests for departure from normality", Biometrika, 60, 613-622

[18] Linear Feedback Shift Registers in Virtex Devices : https://www.xilinx.com/support/documentation/application_notes/xapp210.pdf

[19] G. E. P. Box and Mervin E. Muller, A Note on the Generation of Random Normal Deviates, The Annals of Mathematical Statistics (1958), Vol. 29, No. 2 pp. 610-611

[20] Bresenham: https://it.wikipedia.org/wiki/Algoritmo_della_linea_di_Bresenham

[21] Article - World, View and Projection Transformation Matrices : http://www.codinglabs.net/article_world_view_projection_matrix.aspx

[22] A Parallel Algorithm for Polygon Rasterization Juan Pineda , Apollo Computer Inc. , Chelmsford, iVIA 01824 , juan@apollo.uucp ACM-0-89791-275-6/88/008/0017 (1988)

[23] Barycentric Coordinates : https://it.wikipedia.org/wiki/Coordinate_baricentriche

[24] Reduce Power and Cost by Converting from Floating Point to Fixed Point By: Ambrose Finnerty and Hervé Ratigner https://www.xilinx.com/support/documentation/ white_papers/wp491-floating-to-fixed-point.pdf

[25] DNNDK User Guide: https://www.xilinx.com/support/documentation/ user_guides/ug1327-dnndk-user-guide.pdf

[26] DPU for Convolutional Neural Network v3.0 : https://www.xilinx.com/support/documentation/ ip_documentation/dpu/v3_0/pg338-dpu.pdf

Appendix A

CNN with tensorflow

```
def cnn(features, labels, mode):
  # THIS IS A CNN IN TENSORFLOW
  In_layer = tf.reshape(features["x"], [-1, 28, 28, 1])

  # Convolutional Layer #1
  convolutional_L1 = tf.layers.conv2d(
      inputs=In_layer,
      filters=32,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)

  # Pooling Layer #1
  Pooling_L1 = tf.layers.max_pooling2d(inputs=convolutional_L1,
   pool_size=[2, 2], strides=2)

  # Convolutional Layer #2 and Pooling Layer #2
  convolutional_L2 = tf.layers.conv2d(
      inputs=Pooling_L1,
      filters=64,
      kernel_size=[5, 5],
      padding="same",
      activation=tf.nn.relu)
  Pooling_L2 = tf.layers.max_pooling2d(inputs=convolutional_L2,
   pool_size=[2, 2], strides=2)

  # Dense Layer of neurons 1024
  Pooling_L2_flat = tf.reshape(Pooling_L2, [-1, 7 * 7 * 64])
  dense = tf.layers.dense(inputs=Pooling_L2_flat, units=1024,
   activation=tf.nn.relu)

  # Add dropout for a training
  dropout = tf.layers.dropout(
```

```
31      inputs=dense, rate=0.4, training=mode == tf.estimator.
    ModeKeys.TRAIN)
32
33  # Classification Layer 10
34  Classification = tf.layers.dense(inputs=dropout, units=10)
35
36  likelyhood = {
37      # Generate likelyhood
38      "classes": tf.argmax(input=Classification, axis=1),
39      "probabilities": tf.nn.softmax(Classification, name="
    softmax_tensor")
40  }
```

Listing A.1: CNN example

Appendix B

Second stage full algorithm

```
for each OBJECT : {
  SETUP : {
      load object model
      load scene
      reset depth buffer (z-buffer reset to far)
      load heuristics and Probability
    }
    INIT_POSE : {
      for each SAMPLE {
        do:{
                importance sampling from first stage
          } while (check_pose() == bad)
          Initialize the pose
        }
    }
    EVAL_LOOP :{
    do:{
        INLIER:{
        for each sample{
            transform object
              for each face{
                    load triangle
                    depth evaluation
                    Rasterization  and Depth substitution
                }
              for BB_size {
                  load scene point
                  load depth point (z - buffer)
                  compute inlier
                }
              evaluate weights
            }
          }
```

```
34  COMP_WEIGHTS: {
35      for each sample{
36          calculate final weight
37         }
38      Sort samples
39     }
40  Resample:{
41      Calculate SUM
42      normalize all samples
43      Cumulative sum
44      Resample keeping the most likely
45      Transfer new samples
46    }
47  Diffusion:{
48      for each sample{
49      diffuse randomly
50      }
51    }
52    } while(check_convergency() == bad)
53    } // End evaluation loop
54    AFTER CONVERGECE : {
55      INLIER : { ... }
56      COMP_WEIGHTS  : { ... }
57      }
58  } // End for each object
```

Listing B.1: Complete Second Stage algorithm

Appendix C

Second Stage Code

```
1
2 /*
3  *
4  *
5  * TYPES FOR INITIALIZER
6  *
7  */
8
9 // THESE ARE FOR THE DISTRIBUTION CREATION
10 #define MAX_INIT_BIT   BIT_WEIGHT_FS  // MAX BETWEEN BIT_WEIGHT_FS
      and BIT_SUM_INIT
11 #define BIT_WEIGHT_FS 16  // FOR AXI MUST BE A MULTIPLE OF 8
12
13 ////////////////////////////////
14 //WARNING!!! if you change this you must change also the index of
      the loop
15 // OTHERWISE ENDLESS LOOP
16 #define BIT_SUM_INIT 13 // 13 for the pyramid 4938
17 //WARNING!!!
18 ////////////////////////////////
19
20 #define BIT_WEIGHT_NORM 18 // the truncated version
21
22 #define ONE_TRUNC_INIT 262143 // 2^18 -1
23
24 //256 isthe max  8 bits
25 //1/4938 = 0.000202511 ->  13 bit
26 #define BIT_NORM_COEFF_FS (BIT_SUM_INIT+BIT_WEIGHT_FS)
27
28 // THE PRODUCT OF TWO VARIABLE NEEDS IN EXACT CALUCLUS THE SUM OF
      THE BITS
29 #define BIT_FULL_RES_NORM (BIT_NORM_COEFF_FS + BIT_WEIGHT_FS)
30
```

```
31
32 typedef ap_uint<BIT_WEIGHT_FS> FS_Weight_Data_T;
33 typedef ap_ufixed<BIT_WEIGHT_FS,0>  FS_W_fix_Data_T;
34
35 typedef ap_ufixed<BIT_SUM_INIT + BIT_WEIGHT_FS + 1, BIT_SUM_INIT>
     Full_Res_Sum_Data_T; // need 13 bit to the sum 4938 is bounded
     with 13 bit
36 // I need one extra for the LSB
37 //
38
39 typedef ap_ufixed<BIT_NORM_COEFF_FS, BIT_WEIGHT_FS>
     Norm_coeff_Data_T;
40
41 typedef  ap_ufixed<BIT_FULL_RES_NORM, BIT_WEIGHT_FS>
     Full_Res_Norm_Sum_Data_T;
42
43 typedef  ap_ufixed<BIT_WEIGHT_NORM, 0> Norm_weights_Data_T;
44
45
46 void Init_Crt_Distr(const volatile FS_Weight_Data_T * weights,
47     stream<Norm_weights_Data_T> & Norm_weights) {
48
49   //#pragma HLS INTERFACE m_axi depth=4938 port=weights offset=
     slave bundle=weights
50 //#pragma HLS INTERFACE s_axilite register port=return bundle=
     CRTL_BUS
51
52 #pragma HLS DATAFLOW
53   static stream<FS_W_fix_Data_T> weights2acc_stream;
54 #pragma HLS STREAM variable=weights2acc_stream depth=1
55
56   Init_splitter(weights, weights2acc_stream);
57   Init_load_sum_prob(weights2acc_stream, Norm_weights);
58
59 }
60 /*
61  *
62  * Init_splitter (1)
63  *
64  */
65
66 void Init_splitter(const volatile FS_Weight_Data_T * weights,
67     stream<FS_W_fix_Data_T> & weights2acc_stream) {
68   Init_rd_ram_weights_1: for (u13_Data_T c = 0; c < PYRAMID_PIXELS;
     c++) {
69 #pragma HLS PIPELINE II=1
70     FS_W_fix_Data_T out_fill;
71     FS_Weight_Data_T temp = weights[c];
72     out_fill.range() = temp;
```

```
73      weights2acc_stream.write(out_fill);
74    }
75    Init_rd_ram_weights_2: for (u13_Data_T c = 0; c < PYRAMID_PIXELS;
        c++) {
76  #pragma HLS PIPELINE II=1
77      FS_W_fix_Data_T out_fill;
78      FS_Weight_Data_T temp = weights[c];
79      out_fill.range() = temp;
80      weights2acc_stream.write(out_fill);
81    }
82  }
83
84  /*
85   *
86   * Init_load_sum_prob (2)
87   *
88   */
89
90  void Init_load_sum_prob(stream<FS_W_fix_Data_T> & weights_in_stream
        ,
91      stream<Norm_weights_Data_T> & Norm_weights) {
92
93  #ifdef PRINT_INIT
94    double full_resCalc = 0;
95  #endif
96    Full_Res_Sum_Data_T full_res_sum;
97
98    Initialize_cum_stmem: for (u13_Data_T c = 0; c < PYRAMID_PIXELS;
       c++) {
99  #pragma HLS PIPELINE II=1
100
101     FS_W_fix_Data_T new_weight;
102     weights_in_stream.read(new_weight);
103
104 #ifdef PRINT_INIT
105     std::cout << c << std::endl;
106     if (!c.or_reduce()) {
107       std::cout << "\tw_prec : " << 0 << " \t +" << std::endl;
108     } else {
109       std::cout << "\tw_prec : " << full_res_sum << " \t +" << std
        ::endl;
110     }
111 #endif
112     //with this condition we limits the lower bound of the sum to
        0.0753632 = 2^-16 * PYRAMID_PIXELS
113
114     FS_W_fix_Data_T new_add = new_weight;
115
116     if (!c.or_reduce()) { // if we  are in the first iteration
```

```
117
118 #pragma HLS PIPELINE II=1   // no need of power on initialization
119       full_res_sum = new_add;
120     } else {
121       full_res_sum += new_add;
122     }
123
124 #ifdef PRINT_INIT
125     std::cout << "\tw_new  : " << new_weight << " \t =" << std::
    endl;
126     std::cout << "\t---------------------------- " << std::endl;
127     std::cout << "\tRes fx : " << full_res_sum << std::endl;
128     std::cout << "\tRes ui : " << full_res_sum.range() << std::endl
129         << std::endl;
130     std::cout << "---------------------------- " << std::endl;
131 #endif
132   }
133
134   //256 is the max  8 bits
135   //1/4938 = 0.000202511 ->  13 bit
136   ap_ufixed<BIT_SUM_INIT + MAX_INIT_BIT + 1, MAX_INIT_BIT + 1>
     norm_bef_fill =
137       full_res_sum;
138   norm_bef_fill = hls::recip(norm_bef_fill);
139   // the norm coeff will be 0,13 fixed
140   Norm_coeff_Data_T norm_fill = norm_bef_fill;
141
142   Initialize_Norm: for (u13_Data_T c = 0; c < PYRAMID_PIXELS; c++)
     {
143 #pragma HLS PIPELINE II=1
144
145     FS_W_fix_Data_T new_weight;
146     weights_in_stream.read(new_weight);
147     //with this condition we limits the lower bound of the sum to
    0.0753632 = 2^-16 * PYRAMID_PIXELS
148
149     FS_W_fix_Data_T new_add;
150
151     new_add = new_weight;
152
153     if (!c.or_reduce()) { // if we  are in the first iteration
154                 // no need of power on initialization
155       full_res_sum = new_add;
156     } else {
157       full_res_sum += new_add;
158     }
159
160     Full_Res_Norm_Sum_Data_T full_res_norm = full_res_sum *
    norm_fill;
```

```
161     Norm_weights_Data_T weights2nsgen_fill;
162     if (c == PYRAMID_PIXELS - 1) {
163       weights2nsgen_fill.range() = ONE_TRUNC_INIT;    // set to one
164     } else {
165       weights2nsgen_fill = full_res_norm;
166     }
167 
168     Norm_weights.write(weights2nsgen_fill);
169 
170 #ifdef PRINT_INIT
171     std::cout << c << std::endl;
172     std::cout << "\tw_curr   : " << full_res_sum << " \t *" << std
    ::endl;
173     std::cout << "\tnorm     : " << norm_fill << " \t =" << std::
    endl;
174     std::cout << "\t---------------------------- " << std::endl;
175     std::cout << "\tRes full : " << full_res_norm << std::endl;
176     std::cout << "\tRes trun : " << weights2nsgen_fill << std::endl
    ;
177     std::cout << "---------------------------- " << std::endl;
178 #endif
179   }
180 
181 }
```

Listing C.1: CSN initialiser example