Implementation of the SLAM algorithm on a reconfigurable platform

Υλοποίηση του αλγορίθμου SLAM σε επαναπρογραμματιζόμενη πλατφόρμα ολοκληρωμένων κυκλωμάτων

Pavlos Stoikos

Supervisor: Assoc. Prof. Bellas Nikolaos

2nd committee member: Assoc. Prof. Lalis Spyros

3rdcommittee member: Assoc. Prof. Katsaros Dimitrios

A Thesis submitted for the degree of Diploma Thesis in the Computer Systems Lab (CSL)
Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece

February 2020

Implementation of the SLAM algorithm on a reconfigurable platform

Υλοποίηση του αλγορίθμου SLAM σε επαναπρογραμματιζόμενη πλατφόρμα ολοκληρωμένων κυκλωμάτων

Pavlos Stoikos

Supervisor: Assoc. Prof. Bellas Nikolaos

2nd committee member: Assoc. Prof. Lalis Spyros

3rd committee member: Assoc. Prof. Katsaros Dimitrios

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την ....

....	....	....
Ν. Μπέλλας	Σ. Λάλης	Δ. Κατσαρός
Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής

Dedicated to

my family

Υλοποίηση του αλγορίθμου SLAM σε επαναπρογραμματιζόμενη πλατφόρμα ολοκληρωμένων κυκλωμάτων

Περίληψη

Ο ταυτόχρονος εντοπισμός και η χαρτογράφηση χώρου (SLAM) αφορά την κατασκευή και συνεχή ενημέρωση ενός χάρτη (χρησιμοποιώντας τις πληροφορίες που συλλέγονται από αισθητήρες) ενός άγνωστου περιβάλλοντος, ενώ ταυτόχρονα ορίζει τη θέση ενός ρομπότ μέσα σε αυτό. Ο SLAM είναι ένα πολύ ενεργός ερευνητικός τομέας , καθώς είναι ένας βασικός αλγόριθμος για πολλές τρέχουσες εφαρμογές όπως αυτονομία αυτοκινήτων και πλοήγηση drone. SLAM αλγόριθμοι έχουν χρησιμοποιηθεί πάντα σε ισχυρά υπολογιστικά συστήματα προκειμένου να επιτευχθεί απόδοση πραγματικού χρόνου.Οι διαστάσεις μίας κινητής πλατφόρμας δεν μπορεί να φιλοξενήσει τέτοιους υπολογιστές, επομένως οι ενσωματωμένες πλατφόρμες είναι ίσως η λύση. Παρουσιάζουμε την εφαρμογή του αλγόριθμου KinectFusion σε επαναπρογραμματιζόμενες πλατφόρμες ή πιο συγκεκριμένα, τη Soc Xilinx Zynq-7000 και τη Xilinx Zynq Ultrascale + MpSoc ZCU102. Αυτές οι πλατφόρμες μας δίνουν την ευκαιρία να τρέχουμε διαφορετικά μέρη του αλγορίθμου ταυτόχρονα τόσο στον επεξεργαστή όσο και στην FPGA. Αυτή η έρευνα εξετάζει την απόδοση του αλγορίθμου όταν εκτελείται σε ενσωματωμένα συστήματα και ελέγχει τις δυνατότητες του σε μια επαναρυθμισμένη πλατφόρμα. Η ενσωμάτωση του KinectFusion σε FPGA στην έρευνα αυτή επιτυγχάνει χρόνο εκτέλεσης 0,3816 δευτερολέπτων (2,63 FPS).

Implementation of the SLAM algorithm on a reconfigurable platform

Abstract

Simultaneous localization and mapping (SLAM) concerns the construction and continuous update of a map (by using the information gathered with sensors) of an unknown environment while simultaneously keeping track of a robot's location within it. SLAM is a very active research topic as it is a basic algorithm for many current applications in self-driving cars and in drone navigation.SLAM algorithms have always been used on powerful computing systems in order to achieve real-time performance.The dimensions of mobile platforms cannot host such computers ,consequently embedded platforms are probably the solution.We present an implementation of the KinectFusion algorithm on a re-configurable platform, or to be more exact, the Xilinx Zynq-7000 programmable Soc and the Xilinx Zynq Ultrascale+ MpSoc ZCU102. These platforms give us the opportunity to run different parts of the algorithm simultaneously both on the CPU and the FPGA. This thesis examines the performance of the algorithm when it is executed on these above mentioned boards and checks the potential of it on a re-configurable platform. The FPGA implementation of the KinectFusion in this thesis achieves execution time of 0.3816 seconds (2.63 FPS).

Acknowledgements

First and foremost, I would like to extend my deepest appreciation to my advisor, Prof. Nikolaos Mpellas, for his invaluable support and guidance, which has been instrumental in the development of this thesis. I would like to offer my special thanks to Prof. Spyros Lalis and Prof. Dimitrios Katsaros for their evaluation and valuable comments on my thesis.

I am grateful to my friend Fotis Efstratiadis for our excellent collaboration, while working on this thesis and several projects through the last 2 years

To my friends, thank you for all your support specially during the last year of our studies. We shared experiences and moments that I will never forget.

To my family, thank you for encouraging me in all of my pursuits and inspiring me to follow my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

SLAM is a basic problem for higher-level tasks [1], for example path planning, navigation etc and is widely used in applications like self-driving cars and robotics see Fig.1.1 . SLAM is a key algorithm in localization tasks and an active research field but many challenges lie ahead and one of them is the choice of the platform. Indoor mobile platforms don't have access to powerful batteries considering the limitations they inherently have, so traditional CPUs and GPUs are not an option due to their high power consumption. FPGAs have the benefits of simultaneous computing, power efficiency and the adjust-ability with re-configurable logic, consequently becoming a very promising solution to our problem.

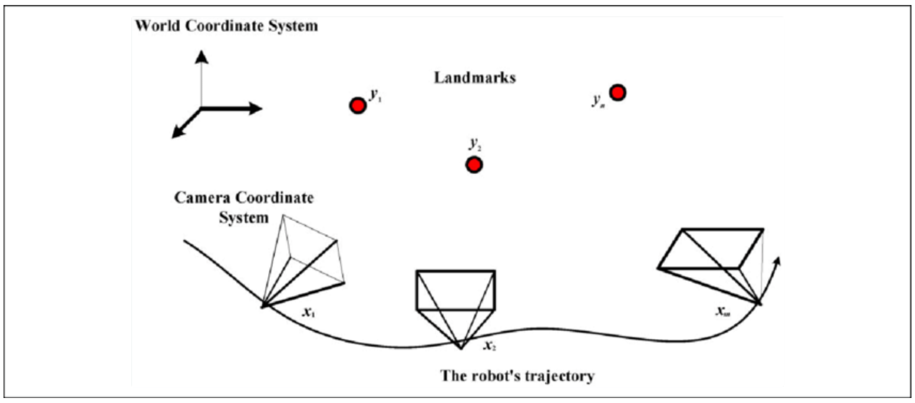


Figure 1.1: Visual SLAM

In recent years, SLAM algorithms have been used on high-performance computers because of their high complexity in order to ensure real-time performance as well as meticulous mapping. The dimensions of the robots do not allow using such powerful computers to run the algorithms. Hence, embedded algorithms are a research topic as they allow efficient implementations and meet real time constraints [2]. There have been many attempts to implement SLAM algorithms on embedded platforms but each implementation heavily depends on which SLAM algorithm you are executing and on which target platform. Although FPGAs look good on paper and seem highly applicable to execute SLAM algorithms, few FPGAs are actually used in commercial products to perform such tasks.

1.2 FPGA

A field programmable gate array [3] is an integrated circuit designed to be reconfigurable by the user which separates them from Application Specific Integrated Circuits (ASIC). They are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects, example in Fig.1.2 , which can be configured by using a hardware description language (HDL). They contain huge resources of logic blocks, and a hierarchy of which can be configured to do combinational logic gates or simple logic gates like NOR and AND. FPGAs can also provide look up tables (LUT), flip flops (FF), Digital Signal processors (DSP), extremely fast I/O rates on buses as well as RAM blocks in order to do digital computations.

It is needed to compare FPGAs, CPUs and ASICs [4] to get a better understanding of the platform. FPGAs and ASICs are different and deep evaluation of the project at hand is needed before choosing one of them. FPGAs can be reprogrammed to do anything we want but ASICs are custom manufactured to do a specific process. On the one hand, we have FPGAs which are more flexible but less efficient and on the other hand, we hold a fixed circuit that performs at a low latency as well as at a low power consumption. Furthermore, when we burn logic right into silicon beforehand, the specific design has better area utilization and performance than the one in the FPGA because of the precision. CPUs and FPGAs handle and

Figure 1.2: FPGA Altera

process data very differently, CPUs execute many processes in parallel and have seen so many architectural changes that give them the edge in so many processes like branch prediction or multi-threading. However, FPGAs with the opportunity to have a more direct approach to hardware give us the chance to achieve better processing performance than a CPU for a fixed algorithm.

1.3 System-on-chic(SoC)

SoC is an integrated circuit [5] that incorporates all the parts of a computer or other electronics. It combines a Processing System (PS) which means a CPU and Programmable Logic (PL) that is the re-configurable logic of a FPGA. SoCs are very common in mobile markets as they consume less power and have excellent performance, as a result of these factors there has been a rising in embedded computing and hardware acceleration .Socs are designed to boost computation load and communication throughput as well as reduce latency for the majority of functions. This is done by offering the user procedures like task scheduling which can improve greatly performance or pipelining that is another fundamental principle for optimizing any process in computer architecture.

1.4 Kinect

Kinect [6] is a motion sensing input device produced by Microsoft. It consists of an RGB camera sensor, a structured light depth sensor that provides the depth of the scene, and a multi-array microphone (not used in our work) as shown in Fig. 1.3 .

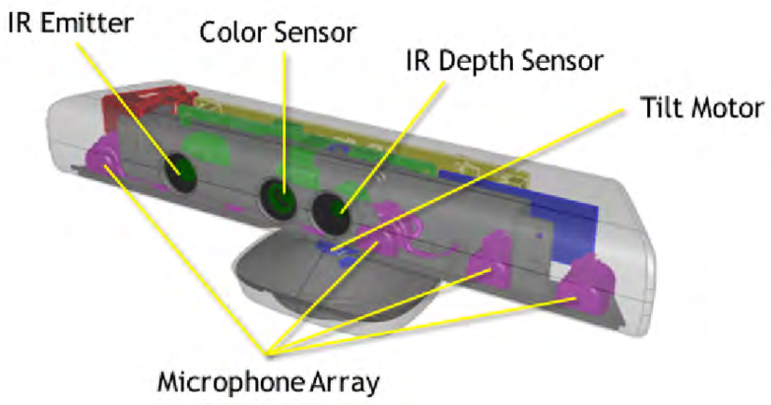


Figure 1.3: Kinect v1

1.5 Contributions

In this thesis, we try to investigate the performance and accuracy of a dense RGB-D SLAM system [7]. SLAMBench [8] provides a KinectFusion [9] implementation in C++ [10], and harnesses the ICL-NUIM dataset of synthetic RGB-D sequences with trajectory and scene ground truth for reliable accuracy comparison of different implementation and algorithms. We investigate the execution time and the accuracy of the KinectFusion executed in the ARM processor of both the ZedBoard [11] and the Xilinx Zynq Ultrascale+ MpSoc [12] [13] which can be seen in Fig. 1.4 and Fig. 1.5 .Furthermore, a high level synthesis approach is examined in order to increase the time execution of the algorithm using the platform's re-configurable hardware.

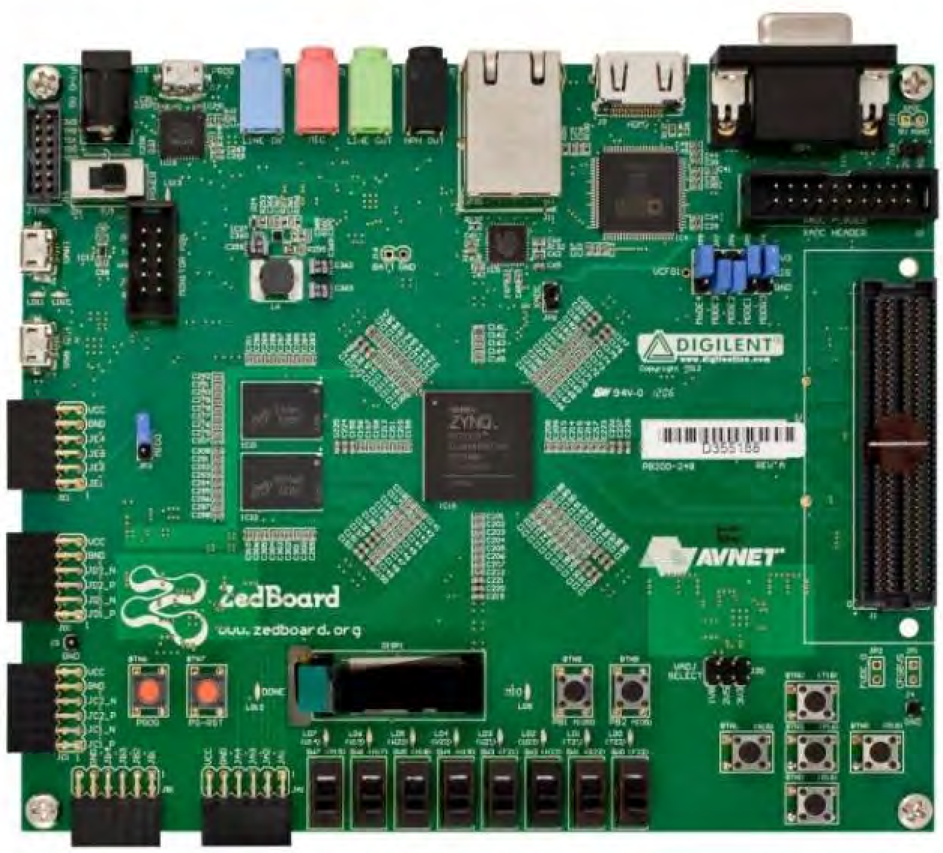


Figure 1.4: Zedboard

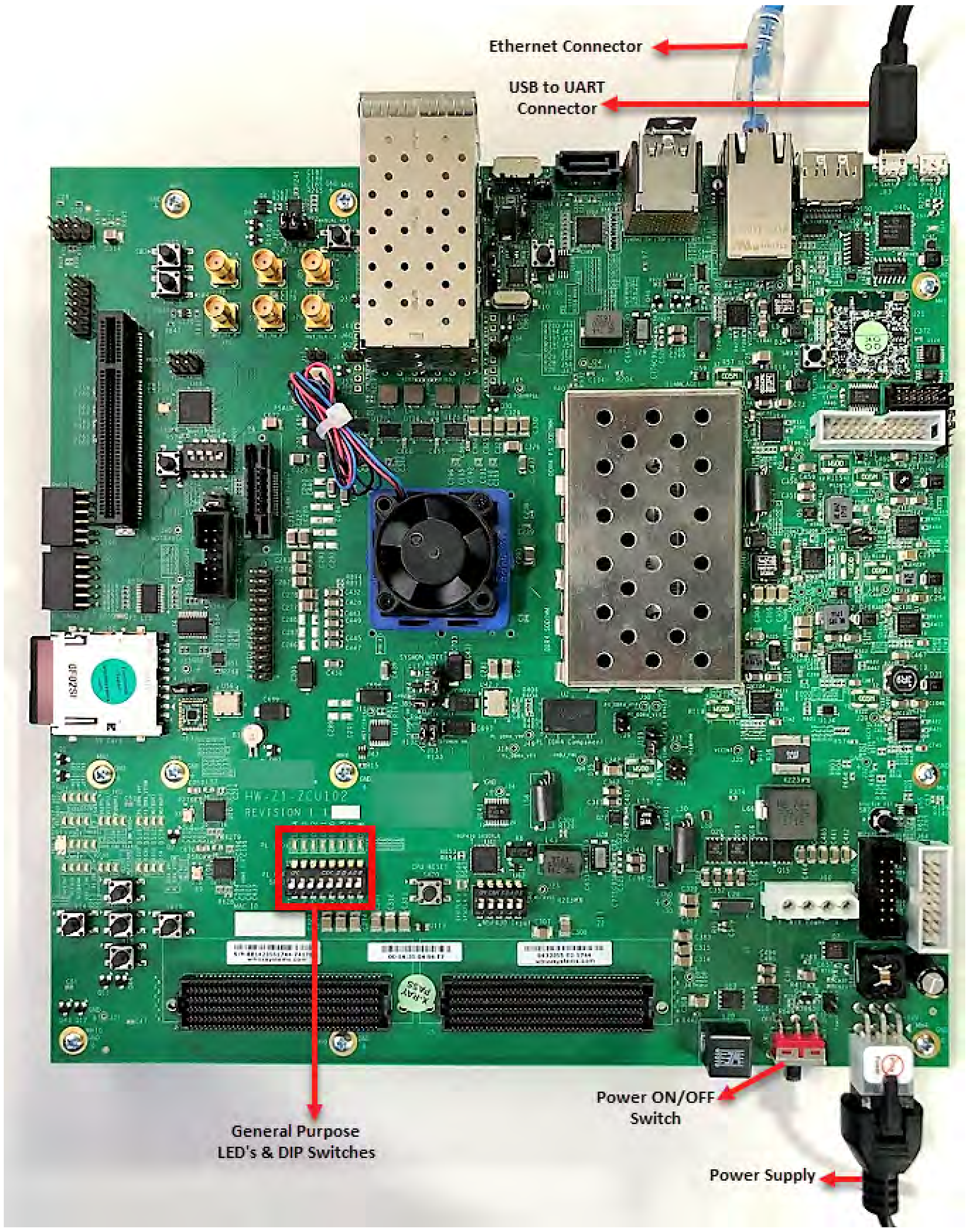


Figure 1.5: ZCU 102

Chapter 2

Simultaneous Localization and Mapping (SLAM)

2.1 SLAM Types

Simultaneous Localization and Mapping is a computer vision and navigation algorithm. A robot must be aware of its position and at the same time construct the environment , which it's what we are trying to achieve. The state of the model can be represented with features or with a volumetric description of the environment. SLAM uses various approaches and is shown through different research papers. For example, maps can store the surroundings as 2D or 3D, and with differing degrees of memory magnitude as well as coarseness. SLAM has not been solved yet as it is stated that more research is needed to achieve realistic results. Implementations of SLAM vary as we have sparse SLAM [14] that it's mostly featured based making it mobile friendly due to its lower complexity but the quality of the output is low. Dense SLAM [15] algorithms produce better results but the computations are costly, so semi dense [16] algorithms have appeared in order to close the gap but they are into an incomplete state. The former describes the quantity of regions used in each received image frame. SLAM algorithms differ also in how the image data are used. Sparse SLAM approaches use a subset of the image frame,while dense ones use almost all the frame. As they handle different amount of pixels their maps vary. On the one hand,the maps from sparse algorithms are point clouds. On the other hand, the maps from dense approaches give us a detailed view of the scene. Furthermore, from the way SLAM algorithms handle information from an image they can be classified as direct or indirect [2]. Indirect SLAM algorithms are the feature-based ones and direct are the the ones that handle the pixels information directly. As the feature extraction is time consuming, direct methods potentially give us the chance to do more computations while being on the same page with indirect ones. However, indirect methods have good tolerance against noisy pixels as they don't handle them directly. There are many well-known SLAM algorithms these days, and by the different ways they handle data we can select the appropriate one for each application or platform.

2.2 Brief Overview of KinectFusion

Kinectfusion [9] is one of the most well-known dense SLAM algorithms. It enables a person to use the Kinect in order to create a 3D reconstruction of an indoor space and performs some steps for each frame:

a) **Acquisition**: An input frame either read from a camera or from a file, in our implementation we use a data-set with a trajectory.

b) **Preprocessing**: Depth values are changed in order to be normalized and then a bilateral filter is used to reduce the noise.

c) **Tracking**: A new pose is build using the Iterative Closest Point algorithm. KinectFusion concludes the difference in the alignment of the current preprocessed depth frame with the depth frame produced from the previous camera pose.

d) **Integration**: The 3D image we have already constructed is changed in order to fuse the aligned data for the frame we have, using the new estimate of the position and orientation which are determined by the tracking phase.

e) **Raycasting**: We ray-trace the 3D map we have constructed and as a result we produce a depth frame from the new pose of the camera.

f) **Rendering**: This phase has no computation value and it is only there for visualizing the 3D surface that we have constructed through all these processes.

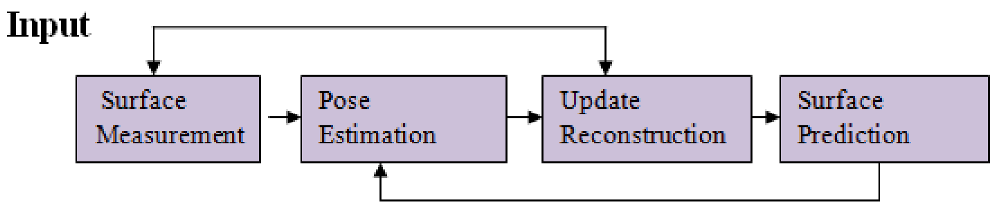


Figure 2.1: General SLAM flow graph

2.3 Brief Overview of SLAMBench

The open-source SLAMBench [8] is a more convenient version of KinectFusion for analysis and bench-marking. The framework [10] is publicly available for a starting point for experimental research in order to check time execution, accuracy and power standards by providing a KinectFusion implementation in C/C++, OpenMP, OpenCL and CUDA as well as an ICL-NUIM data-set of synthetic RGB-D sequences. An overview of the GUI of the algorithm and the second trajectory in the ICL-NUIM data-set running is given in Fig. 2.2.

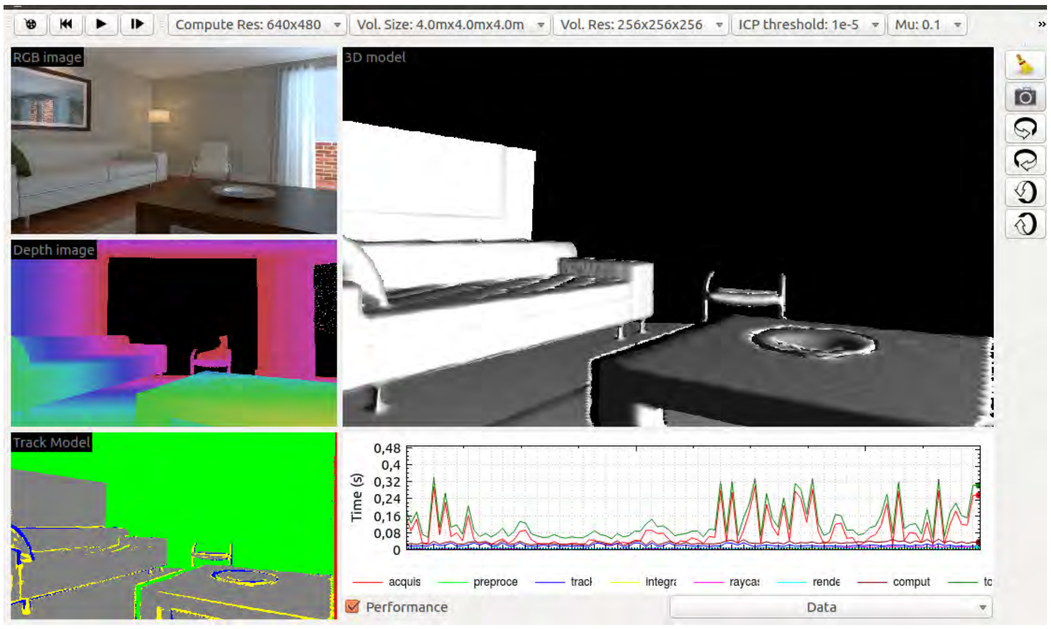

Compute Res: 640x480 | Vol. Size: 4.0mx4.0mx4.0m | Vol. Res: 256x256x256 | ICP threshold: 1e-5 | Mu: 0.1

Figure 2.2: SLAMbench GUI running

2.3.1 Parameters of SLAMBench

SLAMBench offers adjust-ability to certain values like:

a) **Compute-size-ratio**: We decide the resolution of the depth frame that we use as input to the algorithm.

b) **ICP-threshold**: Threshold for the ICP algorithm used in the localization phase.

c) **Integration-rate**: The rate at which the frames that we get are integrated to the indoor scene.

d) **Volume-resolution**: The resolution of the image that we are trying to create.

e) **Tracking-rate:** The rate that we are trying to get a new pose.

f) **Pyramid-levels**: The ICP algorithm does iterations on different levels of the image pyramid, so with this parameter we determine maximum number of them.

These parameters can be changed in order to optimize either execution time or the power consumption but the accuracy of the output should always be acceptable. The instantaneous trajectory error (ITE) is the difference between the actual frame and the computed one. Average trajectory error (ATE) is the average of ITE's of all the frames which are not known until the last frame ends which is the end of the trajectory.

2.3.2 ICL-NUIM DATASET

ICL-NUIM is a data-set which consists of 4 camera trajectories in order to provide us with RGB-D frames. ICL-NUIM can support us with sequences that are of excellent value. They are free of noise, but instead they have noise added to RGB-D frames based on a model that is extracted from Kinect equipment. This noise model is applied to every frame so as to feed the algorithm with realistic frames. This benchmark gives us the chance to produce a lot of pragmatic sequences and provide us with open source code, which we can use to produce our own test data-set.

2.4 SLAMBench Kernels

We hereby present a simple graph of SLAMBench in order to understand the flow of the algorithm as shown in Fig. 2.3. We are going into detail later on and analyze all kernels individually. L stands here for the number of pyramid levels and K is a threshold value. Further down, these values will be explained extensively.

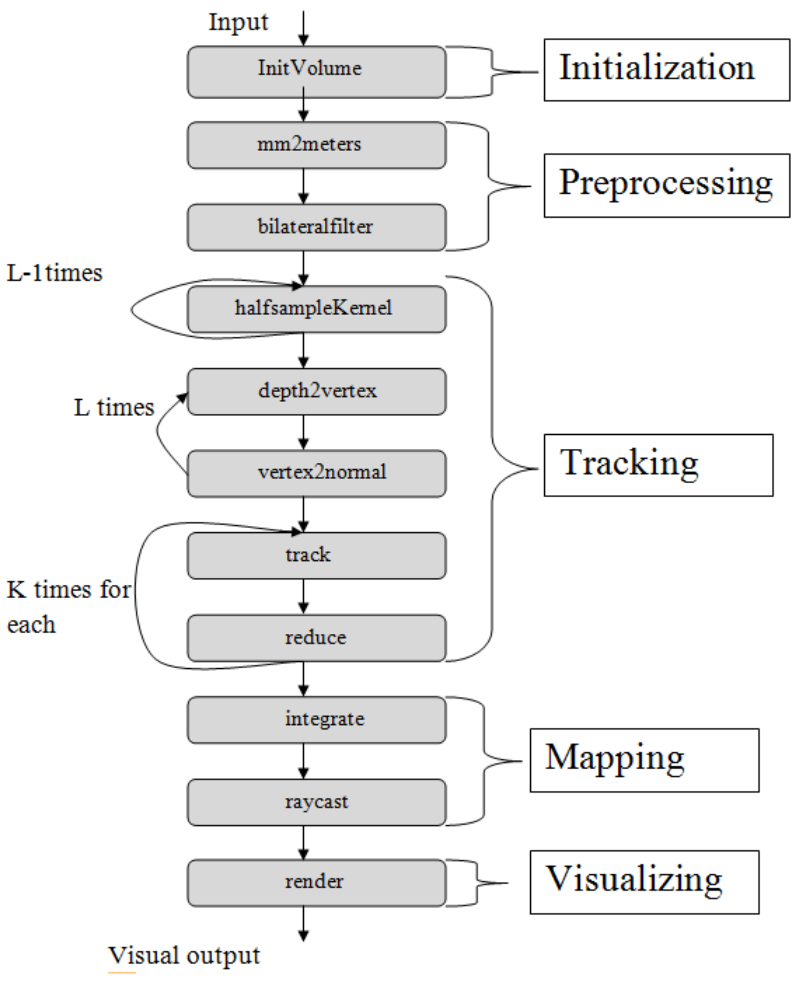


Figure 2.3: SLAMBench Kinectfusion kernels flow graph

SLAMBench has also two kernels but they are only used once at the start, they are part of the initialization process and have no computational cost. To be more specific, one is called generateGaussian that produces a Gaussian bell curve which stores in an array and initVolume that generates the 3D volume. The algorithm also contains many kernels and processes that we need to breakdown in order to understand the flow of the algorithm better.

a) **acquisition**: A new RGB-D frame is acquired. This phase is included in order to measure I/O costs if the frame is acquired from the data-set or from an actual sensor.

b) **mm2meterskernel** re-assembles the output from the sensor and converts it from millimeters to meters. This needs to be done because the frames are given to us in an unsigned short integer format. We need to convert them to floats and resize them so the other kernels can express their values in meters.

c) **bilateralfilterkernel**: The depth map is noisy from the Kinect so we use an edge-preserving filter. This kernel is a filter that changes the values by a weighted average of nearby values so as to reduce the noise or any invalid depth values. This smooths our map and increases its quality at later stages.

d) **halfSamplekernel** creates a three-level image pyramid by taking a sub-sample from the above mentioned filtered depth image. This kernel re-samples the filtered depth map by a factor of two in each level . That means four inputs are mapped to one output.

e) **depth2vertexkernel** transforms the pixel of the new depth image into a 3D point in order to create a point cloud. It multiplies each depth value with a specific matrix as a result an array is generated where the elements give us a 3D Euclidean point.

f) **vertex2normalkernel** produces the normal vectors from the vertex of each point cloud earlier which are then used in the ICP algorithm[17]. The ICP algorithm calculates the distances between two corresponding vertices of the synthetic point cloud and the new point cloud

g) **trackkernel** performs part of the ICP algorithm. It settles a correlation between the synthetic and the new cloud.

h) **reducekernel** calculates the error of the track kernel, it sums up the distances between correlating points in the input and the predicted cloud.

i) **updateposekernel** generates a new pose from the reduced kernel output.

j) **checkposekernel** verifies the output of the reduction process and resets the pose to an old stable one if needed.

k) **integratekernel** integrates the new depth map into the current constructing 3D map, it iterates over the volume to make updates to every element.The volume is made of Truncated Signed Distance Function (TSDF) [18] values. Every element has a weight which represents the certainty of the surface measurement at that specific position on the map.Integration iterates over the whole 3D reference weight map in order to update every element. Updating means computing an average of the TSDF value we already have and the new TSDF value. Values are positive in front of the surface and negative behind. Numbers between 0 and 1 are at the surface of the object.

l) **raycastkernel** produces a vertex and a map by making a ray from each pixel into the 3D Volume from the current pose estimate. It makes a prediction of the how the input arrays should look like if the view point of the camera made the observation from this specific position see Fig 2.4 and algorithm 1.

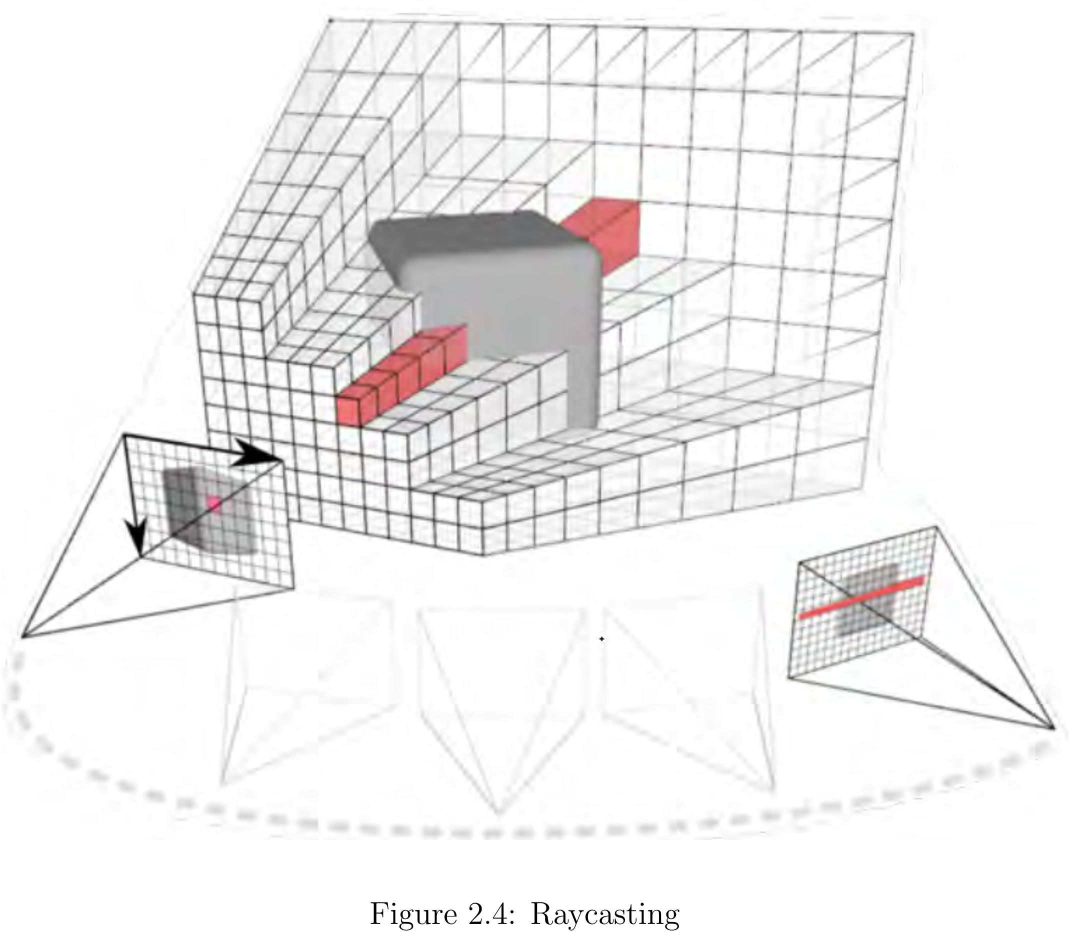


Figure 2.4: Raycasting

Algorithm 1 Raycasting pseudocode

```
1: Procedure Raycasting
2: for each each pixel u ∈ pose do
3:     $ray^{start} \leftarrow project(nearPlane)$ //Initiate ray
4:     $ray^{end} \leftarrow project(farplane)$
5:     $ray^{dir} \leftarrow ray^{start} - ray^{end}$
6:     $ray^{len} \leftarrow 0$
7:     while $ray^{len}$ within volume bounds do //Execute raycast
8:         $ray^{len} \leftarrow ray^{len} + 1$
9:         $g \leftarrow$ first voxel among $ray^{dir}$
10:        if $F_{trilinear}(g_x, g_y, g_z) == 0$ then // If g interpolates a polygon
11:            $p \leftarrow$ extract trilinear interpolated grid position
12:            $vertex(u) \leftarrow$ convert p from grid to 3D position
13:            $normal(u) \leftarrow$ extract surface from gradient as $\nabla$ p
```

m) **renderDepthkernel** visualizes the depth map via color coding.

n) **renderTrackkernel** visualizes the tracking by picking different colors for each pixel.

o) **renderVolumekernel** visualizes the 3D volume we have reconstructed.

Chapter 3

SLAM Implementation and Experimental Evaluation

The trajectory used in the timing profile is provided by the SLAMBench [10] platform. The ICL-NUIM data-set has 4 trajectories and we pick the second one which contains 882 frames with all the information needed. With the use of the data-set we can easily check the accuracy of our algorithm because we have a ground truth trajectory file. We check the accuracy of the tracking compared to the ground truth trajectory via the above mentioned file. The parameters used to run SLAMBench are the following:

-s 4.8: volume size

-p 0.34, 0.5, 0.24: init-pose

-z 4: rendering rate

-c 2: compute size ratio

-r 1: integration rate

-k 481.2, 480,320,240: camera

They are plenty of parameters to change but the other ones are set to default, for example volume resolution is 256,256,256.

3.1 CPU x86-64 timing results

With the above mentioned parameters the implementation of the Kinectfusion on a laptop with a CPU (i7-6500U @ 2.50 GHz) is shown in Fig. 3.1 and based on the time execution on each process of the algorithm we can see where load is.

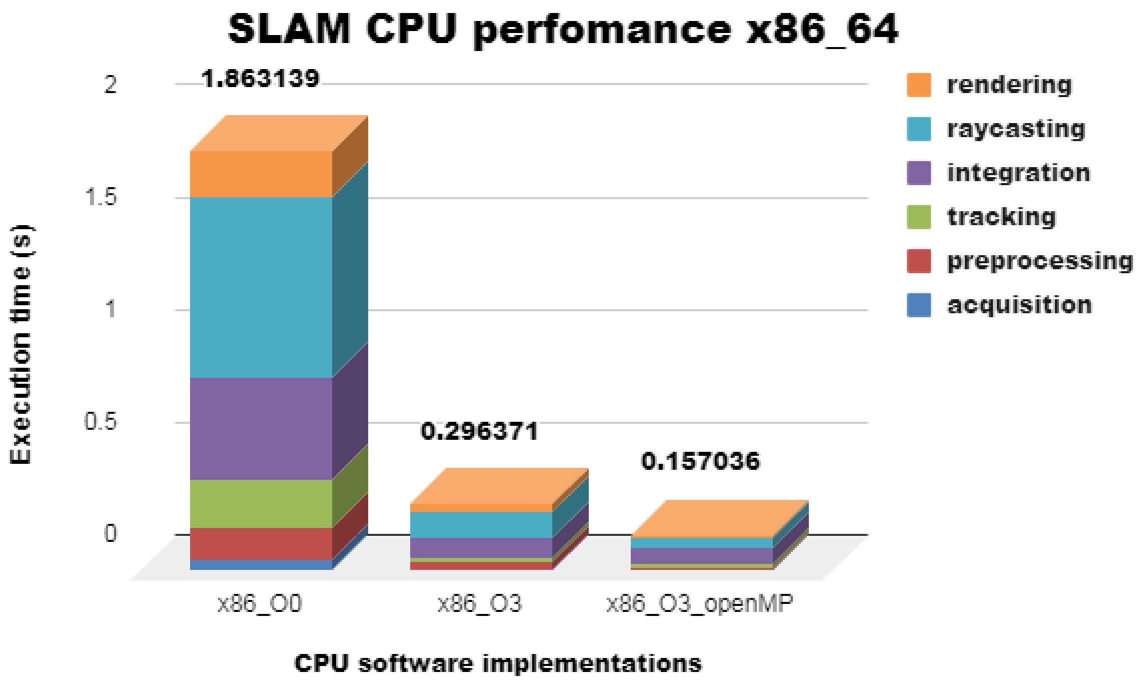


Figure 3.1: SLAM CPU x86 software implementations

3.2 FPGA System-on-Chip (MPSoC)

The implementation time and the computation load on the Arm processor of Zynq Ultrascale+Mpsoc) are in Fig. 3.2 and it seems the load is divided similarly to the x86-64 processor.

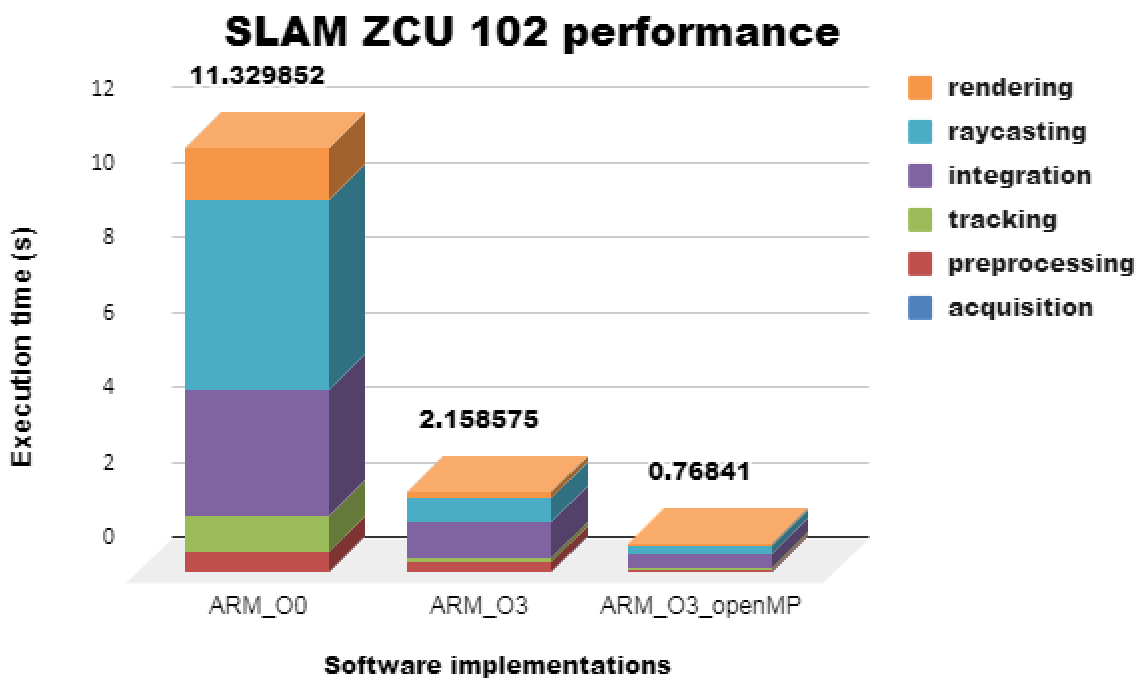


Figure 3.2: SLAM ZCU 102 software implementations

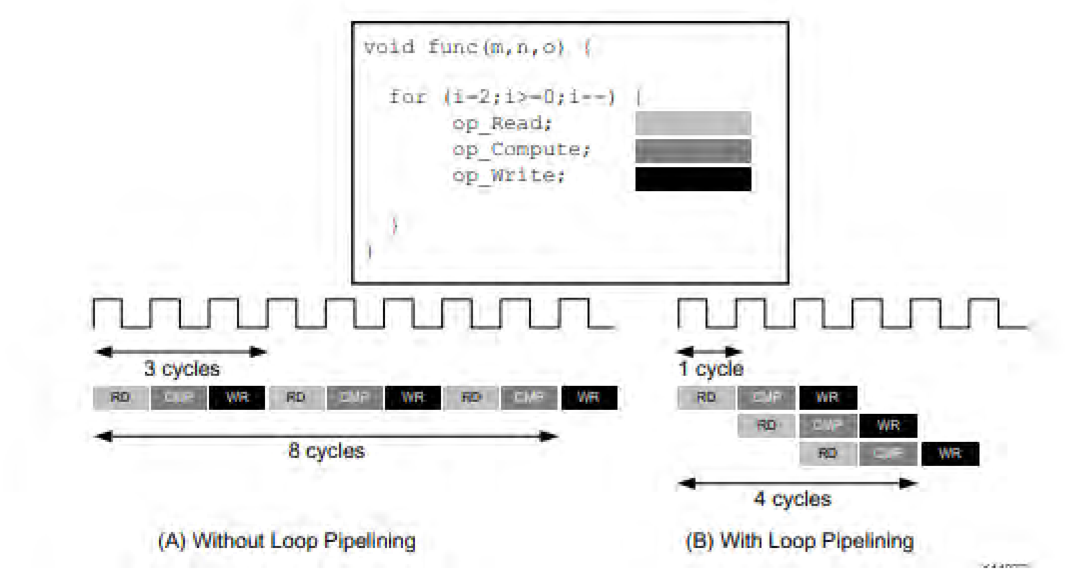
3.3 Implementation of the Mm2 to Meters conversion on HLS

Based on the information given by the graph in Fig. 3.14 we need to use the FPGA to speed up [19] the preprocessing functions. The preprocessing consists of two processes one is the mm2meterskernel. It passes through an amount of pixels and divides them by 1000.

a) **Code**: The mm2meterskernel function was copied straight from the current implementation. Some changes to the pointers were made in order for the code to be synthesizable and we removed the omp pragma.

b) **Interface**: We use the s-axilite mode [20] which specifies an AXI4-Lite slave I/O protocol and the m-axi mode that specifies an AXI4 master I/O protocol. The return is specified as an AXI4-Lite interface and all the types of argument except arrays. All arguments are grouped them to the same AXI4-Lite interface. With the AXI4 master interface we specify arrays and pointers. Also, Vivado HLS produces an-associated set of C driver files during the Export RTL process. We make use of the memcpy [21] function to transfer data to or from a top-level function argument specified with an AXI4 master interface.

c) **Pipelining**: The body of the loop achieved initiation interval of II=1 [22]. Pipelining is used to improve throughput. Pipelining offers us simultaneous operations: each execution step does not have to complete all operations before it begins the next operation Fig. 3.3. The iteration latency stays the same but we should expect a speed-up compared to the original implementation. We make use of the burst mode using the memcpy[21] function. The mm2meterskernel contains two loops, in the current implementation, based on the parameters we gave the outer one is of size 240 and the other 320. We tried per iteration of the outer loop to read a burst of 640 shorts and write 320 floats, as the inner loop to complete all iterations requires as input these values and produces the 320 floats. It turns out this is not the best approach. The optimal choice is to read one value and write back one per iteration of the loop.



```
void func(m,n,o) {

  for (i=2;i>=0;i--) {
        op_Read;
        op_Compute;
        op_Write;

  }
}
```

Figure 3.3: Loop pipelining

	Latency	Iteration Latency
Mm2metersKernel(1 read 1 write option)	318036	1325
Mm2metersKernel(640 reads 320 writes option)	76838	40

Table 3.1: Mmm2meterskernel read/write options

As we can see from Table 3.1 generated from the Vivado HLS [20] we can choose the optimal implementation.

3.4 Implementation of the Bilateral Filter on HLS

The other function that is part of the preprocessing is the bilateralfilterkernel, which is an application of a certain filter.

a) **Padding**: To achieve the bilateralfilterkernel its purpose it uses a certain edge-preserving filter. In the current implementation of the kernel makes use of clamp [23]. Unfortunately, the FPGA unlike the CPU is not good at branch prediction so another approach need to be considered. The initial approach uses a 5x5 window for its computations and clamp handles the edges so that we don't have a problem with bounds. So instead of this, we introduce padding as a solution.

The bilateral filter consists of two filters, one is a space filter which aims to reduce noise and blur out everything without preserving the edges. The other one is a range filter which depends on the context image by smoothing differences in intensities. As a result, pixels with intensity values similar to that of central pixel are considered for blurring, while others that differ are maintained Fig. 3.4.

Hereby we present a mathematical description of the bilateralfiterkernel Fig. 3.5:

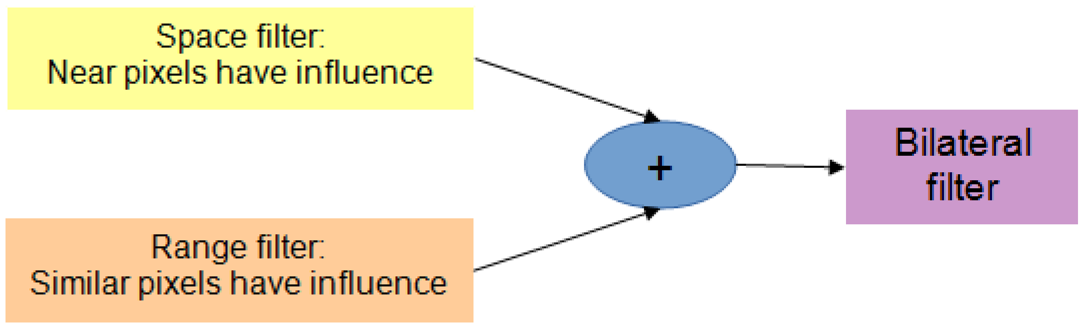


Figure 3.4: Bilateral filter flow graph

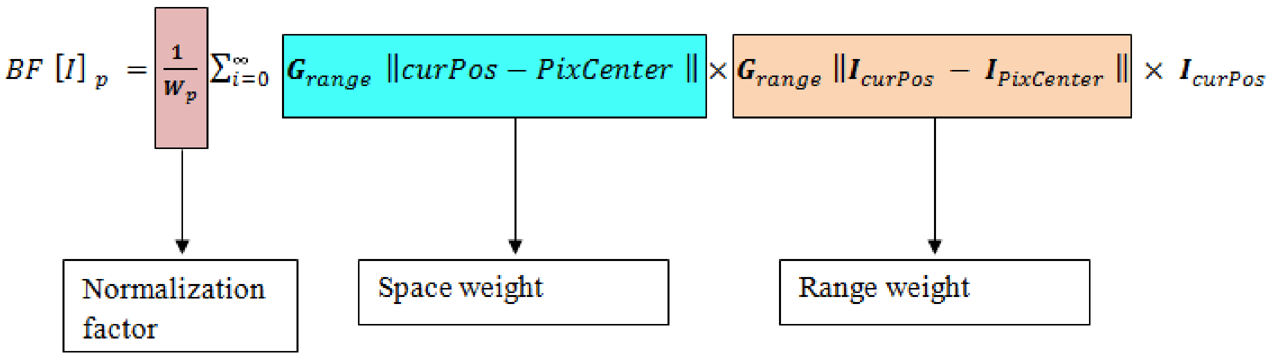

$BF[I]_p = \frac{1}{W_p} \sum_{i=0}^{\infty} G_{range} \|curPos - PixCenter\| \times G_{range} \|I_{curPos} - I_{PixCenter}\| \times I_{curPos}$
Normalization factor
Space weight
Range weight

Figure 3.5: A mathematical description of the bilateralfiterkernel

The depth array which is the input is padded as shown in Fig. 3.6 . The padding of the input is executed on the Arm processor.

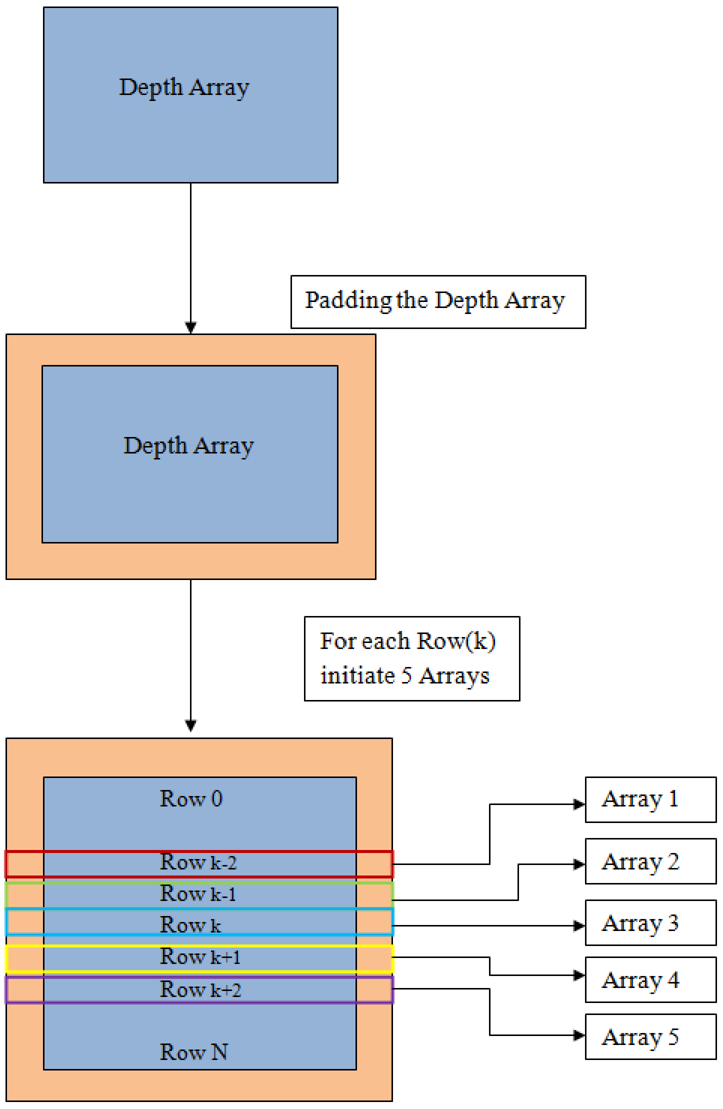
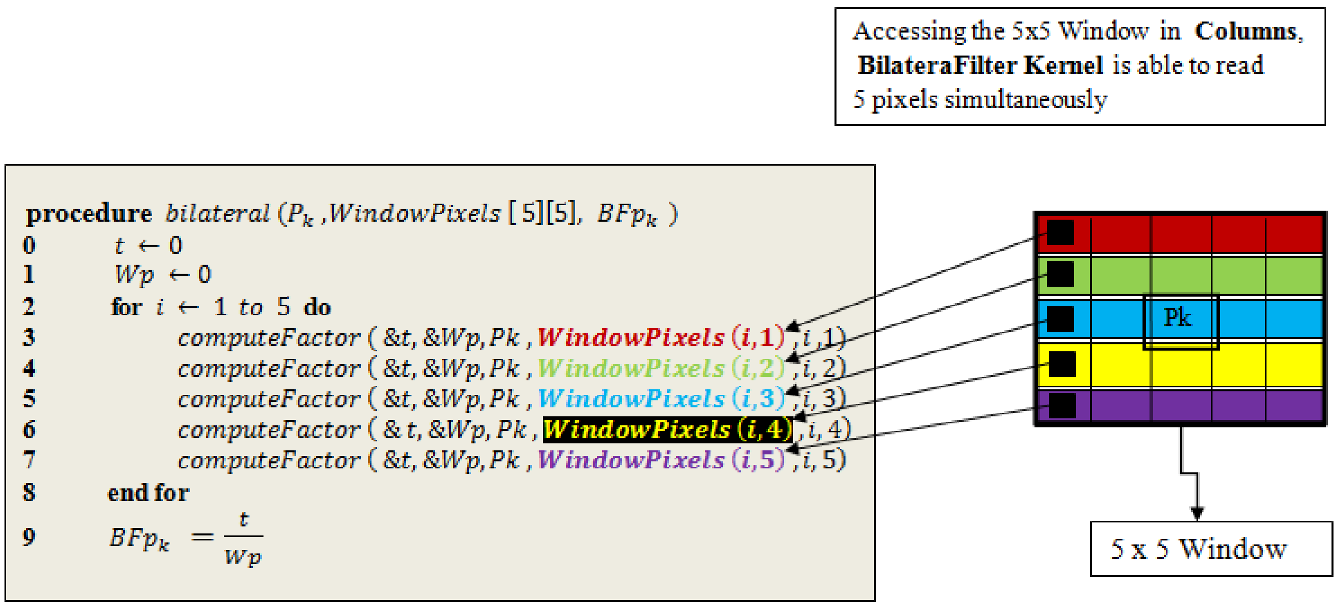


Figure 3.6: Input padding

We can demonstrate the bilaterafilerkernel processes in pseudo code together with the padding as to make it easier to understand. The algorithm 2 demonstrates the bilateralfilterkernel function combined with Fig. 3.7 which shows the processing of the 5x5 window and in algorithm 3 the computations for the pixels are done.

Algorithm 2 Bilateralfilterkernel pseudocode

```
Procedure bilateralfilterkernel (pad-depth,out)
for each row k from pad-depth do
    Copy 5 rows from pad-depth to 5 different arrays
    Copy 2 rows above the row k (k-1,k-2)
    Copy 2 rows below the row k (k+1,k+2)
    Copy the row k
    for each Pixel($P_k$) on row k do
        Initiate WindowPixels
        bilateral($P_k$, WindowPixels [5][5],BF$_{Pk}$)
    end for
end for=0
```



```
procedure bilateral ($P_k$ ,WindowPixels [5][5], $BFp_k$ )
    $t \leftarrow 0$
    $Wp \leftarrow 0$
    for $i \leftarrow 1$ to 5 do
        computeFactor ( &t, &Wp, Pk , WindowPixels (i,1) ,i ,1)
        computeFactor ( &t, &Wp, Pk , WindowPixels (i,2) ,i, 2)
        computeFactor ( &t, &Wp, Pk , WindowPixels (i,3) ,i, 3)
        computeFactor ( & t, &Wp, Pk , WindowPixels (i,4) ,i, 4)
        computeFactor ( &t, &Wp, Pk , WindowPixels (i,5) ,i, 5)
    end for
    $BFp_k = \frac{t}{Wp}$
```

Accessing the 5x5 Window in **Columns**, **BilateraFilter Kernel** is able to read 5 pixels simultaneously
Pk
5 x 5 Window

Figure 3.7: Bilateral made up function pseudocode

Algorithm 3 Window processing of bilateral

```
Procedure computeFactor (t,$W_p$,PixelCenter,CurPos,i,j)
$factor = Gspace(i,j) * Grange(||CurPos - PixelCenter||)$
$t+ = factor * CurPos$
$Wp+ = Wp$
```

b) **Interface**: We added a combination of an AXI4-Lite slave I/O protocol and an AXI4 master I/O protocol. Set max-read-burst-length=256 for input and max-write-burst=256 for output.This burst mode performs data transfers, the interface indicates the base address and the size of the transfer. The data samples are then transferred in consecutive cycles.

c) **Pipelining**: The bilateralfilterkernel consists of four nested loops. The first as well as the second iterate over the whole computation size of the image (240*320 based on our parameters) and the two following loops are the window size of the filter. Pipelining directive was used at the second one. The initial implementation achieved iteration interval (II) equal to 25 because of the clamp function and the double nested loop inside. After getting rid of clamp and rewriting the process with a better approach we managed to drop it down to II=13.

The problem is now that in the "window" loops we are unable to schedule all the loads and get the values we want because a memory core has limited ports. We cannot read from the same array all the values we want at the same time so we have to wait 13 cycles because of the few ports. As a result, we manually unroll the five iterations of the last loop and split the information to five different arrays so we can access more information simultaneously now that we have more reading ports to work with. With this implementation we achieve II=3. Unfortunately, to achieve II=1 we need to do complete portioning to all 5 arrays but we sacrifice a lot of resources to do that and it's not worth the trade.

The other option was to do complete partitioning to the original array before the manual unrolling but the Vivado HLS cannot partition such a big buffer. Consequently, the only option is to partition the 5 little ones but we do not get extremely better results compared to the amount of area that will be provided. In addition, the bilateralfilterkernel does not take so much time to execute compared to other processes in the algorithm, e.g. integratekernel, in order to be logical to provide such resources, so the implementation with II=3 is the optimal choice.

d) **Memcpy**: Memcpy as we discussed previously is used to transfer data to or from a top-level function argument specified with an AXI4 master interface. Initially, we read one value from each array, as we have placed the memcpy inside the double nested loop and because of it we also write back only one. Per iteration of the double nested loop we had one load and one store with exception to the Gaussian array. The Gaussian array accesses exist in the last loop, so we read values at each iteration of the final nested loop which means the loads are considerably more. In the final approach, we use the memcpy function to load the Gaussian array before the loops execution as the values remain constant. The size of the array is extremely small (5 floats) and we can afford to read it all at the beginning. Additionally, at the start of the second nested loop, each time we load the 5 arrays we need with new values and at the finish of it we write them back the at output buffer.

e) **Expf function**: Two implementations of expf were used exp1 and exp2 [24] in order to reduce the cycles of the expf. The approximate approach of these functions will reduce the computation load of the process compared to expf. Unfortunately, none of them gave better execution time so they were not implemented. We have to note also that these functions did not cause the accuracy to drop.

f) **Fabric clock frequency**: The final implementation of the bilateralfilterkernel was tested at 100,250 and 300 MHz .

We can see the resources we have to give for the mm2meterskernel and the bilateralfilterkernel in Table 3.2 and in Table 3.3

The first implementation was done with both kernels not optimized. Afterwards, we checked the time execution with mm2meterskernel on FPGA and bilateralfilterkernel on CPU, as well as vice versa. This test procedure was done later on with these kernels fully optimized. At first we saw that mm2meterskernel had relatively the same execution time on the FPGA and on the CPU when completely developed. We discovered that mm2meterskernel does not affect the time execution of the preprocessing and it will be a better choice to leave it to the CPU. It's not worth providing resources to it.

As we can see in Fig. 3.8 when all the processes are running on ARM the mm2meterskernel does not affect the preprocessing and we should focus implementing only the bilateralfilterkernel.

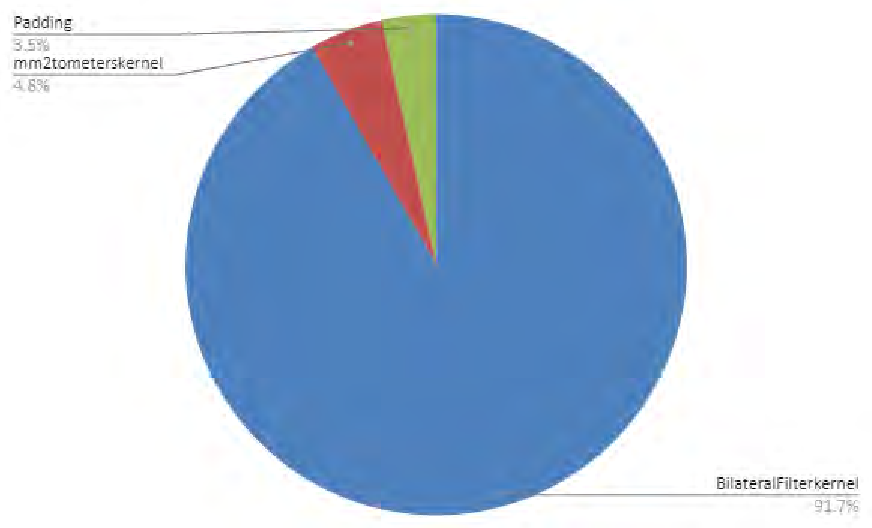


Figure 3.8: Initial computation load of preprocessing functions

As a result the final implementation reached a time execution of 0.0084s, see Fig.3.10 and Fig. 3.17, and the load has seen a dramatic change Fig. 3.9

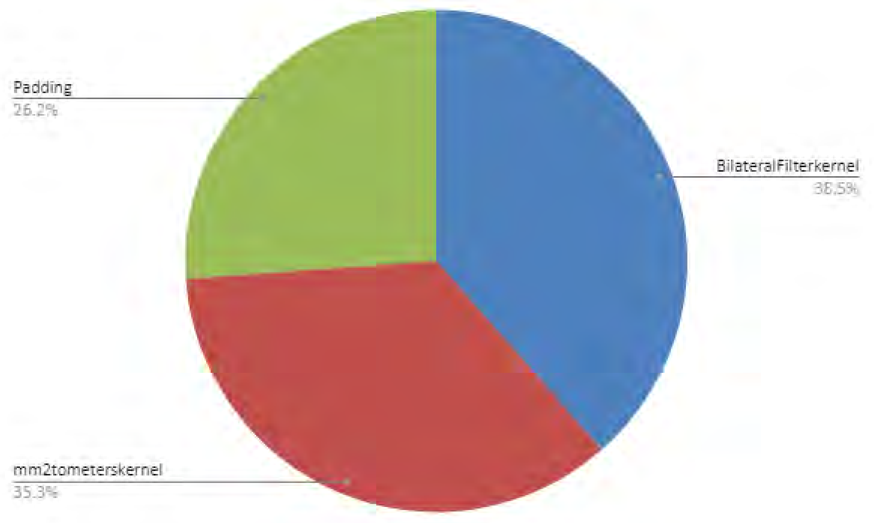


Figure 3.9: Final computation load of preprocessing functions

So we have reached a point where computation time of the preprocessing is 2x faster than the x86-64 CPU and all of its functions are on equal footing Fig. 3.9. Consequently, in order to lower the implementation time even more, mm2metereskernels needs to be revised and find a new approach. Furthermore, bilateralfilterkernel supports parallelism which means we can implement 2 IPs in our design to increase performance even more. However, as preprocessing takes up only 10 percent,which means is the third most time consuming kernel , of the whole algorithm it's not advised to give up more resources of the FPGA. This is a good point to draw the line, considering the time execution, and start implementing other kernels.

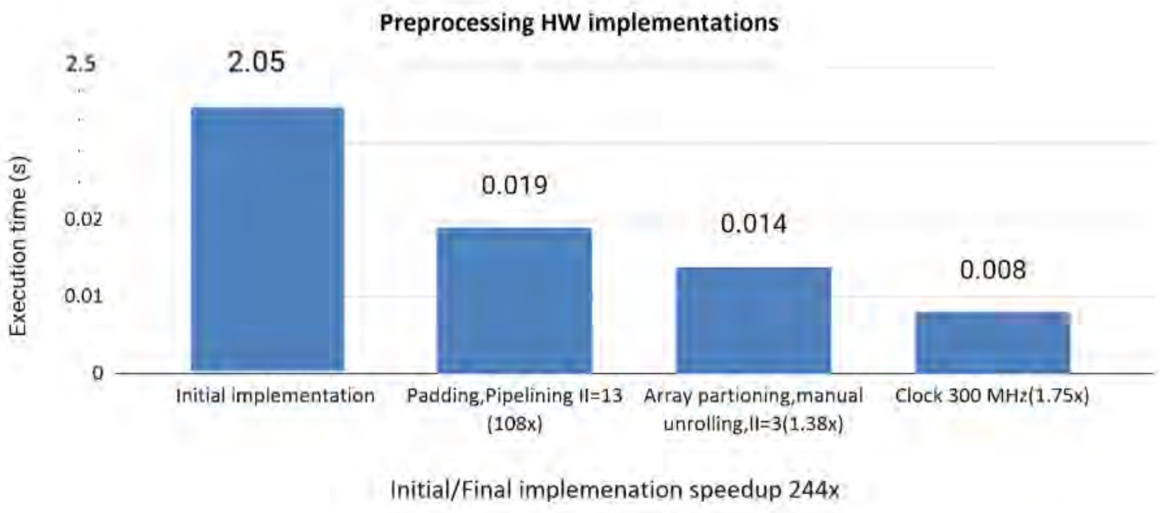


Figure 3.10: Preprocessing HW implementations

3.5 Implementation of the Integration step on HLS

The integrate step does volumetric integration and goes through all the 3D volume, as a result such a heavy computational function Fig.3.14 should be implemented in hardware.

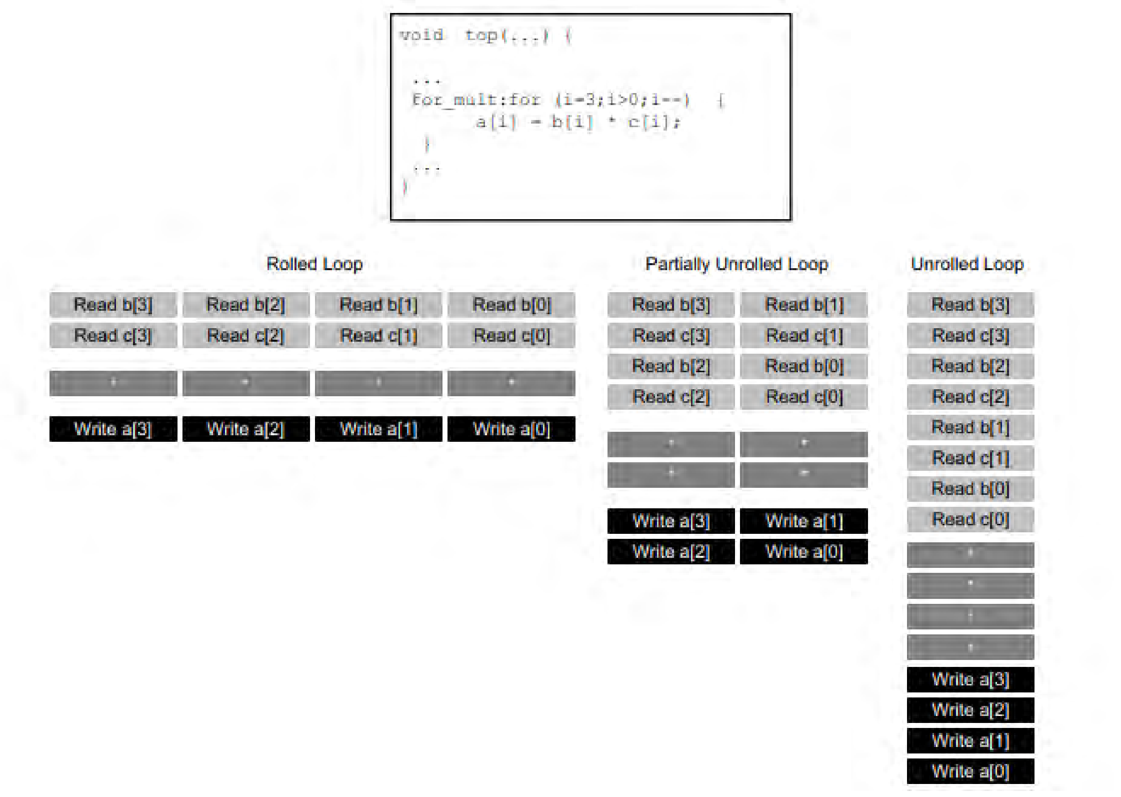
a) **Interface**: A combination of an AXI4-Lite slave I/O protocol and an AXI4 master I/O protocol makes up the interface . Max-read-burst=256 and max-write-burst=256 were set for the input/output 3D volume bus.

b) **Loop interchange**: To make the code able to be pipelined we had to change the loops from (y,x,z) to (z,x,y). We need to loop over every element in the 3D volume in a specific way. With (y,x,z) for the complete iteration of the z loop we had to load via memcpy almost all the 3D Volume which is not possible. Consequently, we changed it to (z,x,y) as it goes through the 3D volume the way we need. For each completion of the y loop we only need 256 elements, as is the default value of the volume size of the dimension, compared to the previous implementation

c) **Memcpy**: Firstly, we read two values from two arrays per iteration of the triple nested loop (3D Volume and depth image), as we have placed the memcpy inside the loops. Consequently, we wrote back only one at the 3D volume. However, by interchanging the loop we decided to change that. We load 256 values before the start of the third loop and we write back 256 once it has finished. We also moved the load of the depth value from inside the triple nested loop to the start of the integration step. The FPGA is big enough that gives us this privilege of reading the whole array at the start, in our case a size of [320*240], before the iterations begin.

d) **Pipelining**: After the loop interchange the code was not ready to be pipelined . We had also to change the memcpy as we mentioned above. The body of the loop achieved initiation interval of II=2 when the memcpy of the depth array was inside the loops, afterwards the initiation interval achieved II=1.

e) **Loop unrolling**: Unroll the inside loop by a factor of 2 [22]. Pragma HLS unroll, generates multiple copies of the loop , so that we can exploit the parallel architecture Fig. 3.11.



```
void  top(...) {

 ...
 For_mult:for (i=3;i>0;i--)  {
       a[i] = b[i] * c[i];
  }
 ...
}
```

Figure 3.11: Loop unrolling

f) **Array partition**: Cyclic Partitioning of the input volume array by a factor of 2 to make use of the loop unrolling[22]. Vivado HLS provides pragmas to increase local memory bandwidth. By partitioning arrays as in Fig. 3.12 we increase the number of load/stores ports.

Vivado HLS provides three types of array partitioning:

• **block**:The original array is split into equally sized blocks of consecutive elements of the original array.

• **cyclic**:The original array is split into equally sized blocks interleaving the elements of the original array.

• **complete**:The default operation is to split the array into its individual elements. This corresponds to implementing an array as a collection of registers rather than as a memory.

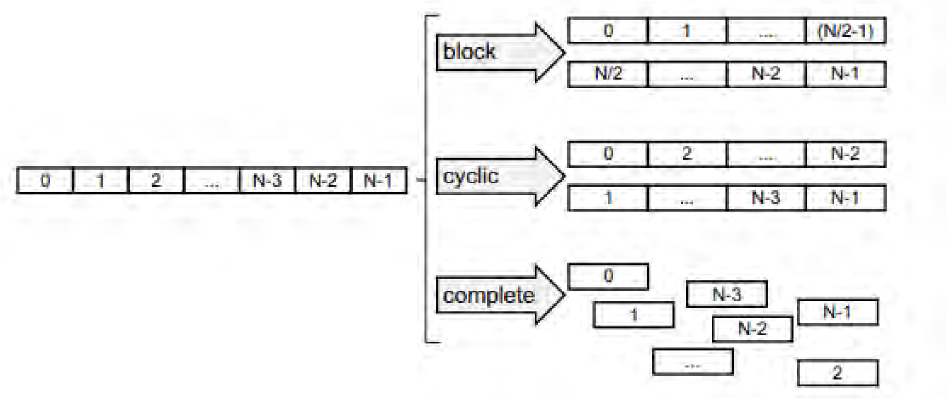


Figure 3.12: Array partitioning

g) **Fixed point arithmetic**: Floating point operations are expensive on a FPGA and an option is to use fixed-point numbers [25], where the decimal point is placed at a fixed position within the bit representation. The selection between these data type depends on operations of each kernel, usually fixed point numbers results in a more efficient design. However, we accelerate only portions of the algorithm, conversions from floating point numbers to fixed point numbers will have to be taken into consideration .This is because KinectFusion will run on an ARM, where floating point operations will be used. Another issue is the accuracy that the fixed point operations offer. Floating point numbers are extremely accurate at both small and large values, while fixed-point figures are only usable within a small range because of a fixed step size and a low maximum value. After implementing Xilinx's [26] fixed point arithmetic the results were disappointing. The limitation of the fixed point precision led to controversial results regarding the accuracy of the algorithm. Furthermore, the time execution of the integration step increased possibly due to many data type conversions. In consequence, we decided to keep the floating point operations.

h) **Bipartite tables** [27]: In the integratekernel the square root (sqrt) function has a heavy computation load. So instead we use symmetric bipartite tables for accurate approximation of the function. Unfortunately, our implementation didn't give promising results as the performance of the integratekernel dropped and this approach was dropped also.

i) **Parallelism**: Due to the nature of the integratekernel we can use multiple accelerators to increase the performance of this section of the algorithm. In this implementation they are used 4 IP's (integratekernel) to speed up the process.

j) **Fabric clock frequency**: The final implementation of the integratekernel was tested at 100, 250 and 300 MHz.

We can see the demand in resources in Table 3.2 and in Table 3.3 .

Finally, we have reached a computation time of the integration step equal to 0.064144 seconds see Fig. 3.13 and Fig. 3.17 .

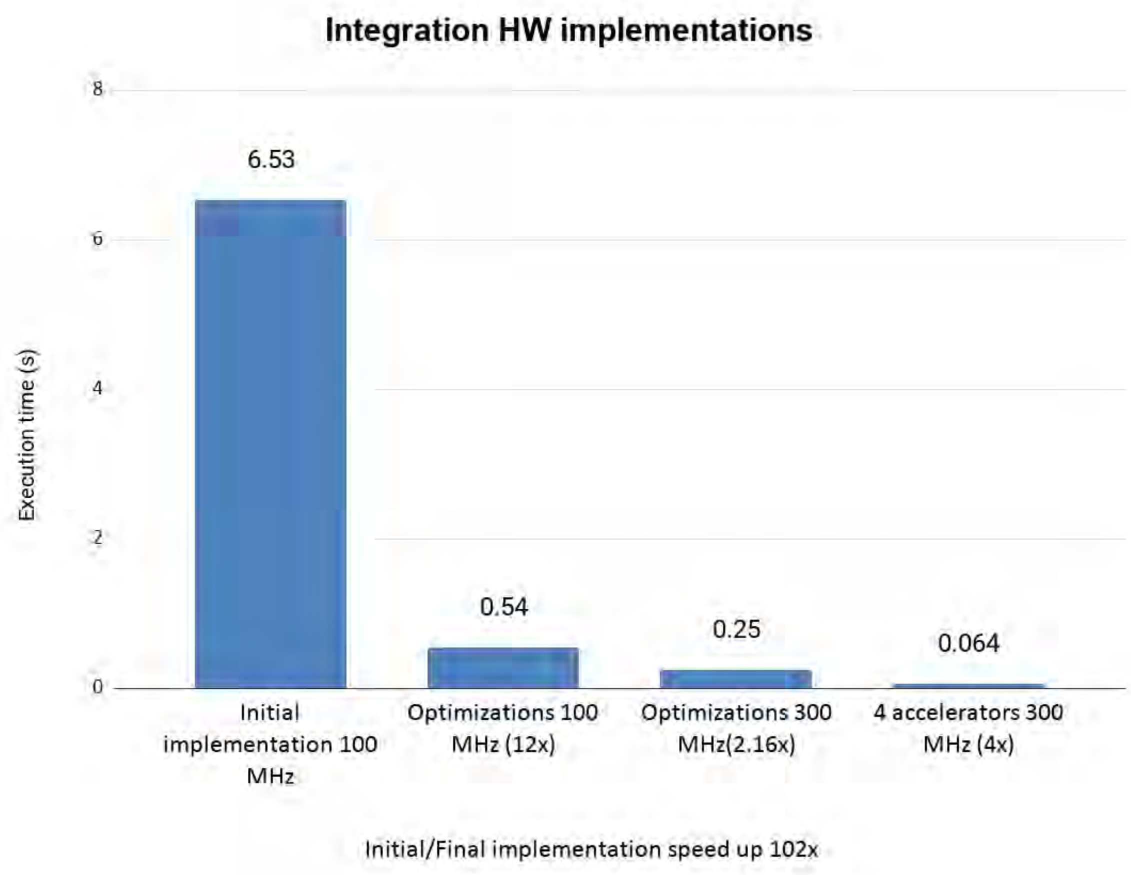


Figure 3.13: Integration HW implementations

3.6 Implementation of the Raycasting step on HLS

The raysasting step is the second most time consuming process Fig. 3.14, after the integratekernel. Therefore, the function should be implemented in hardware.

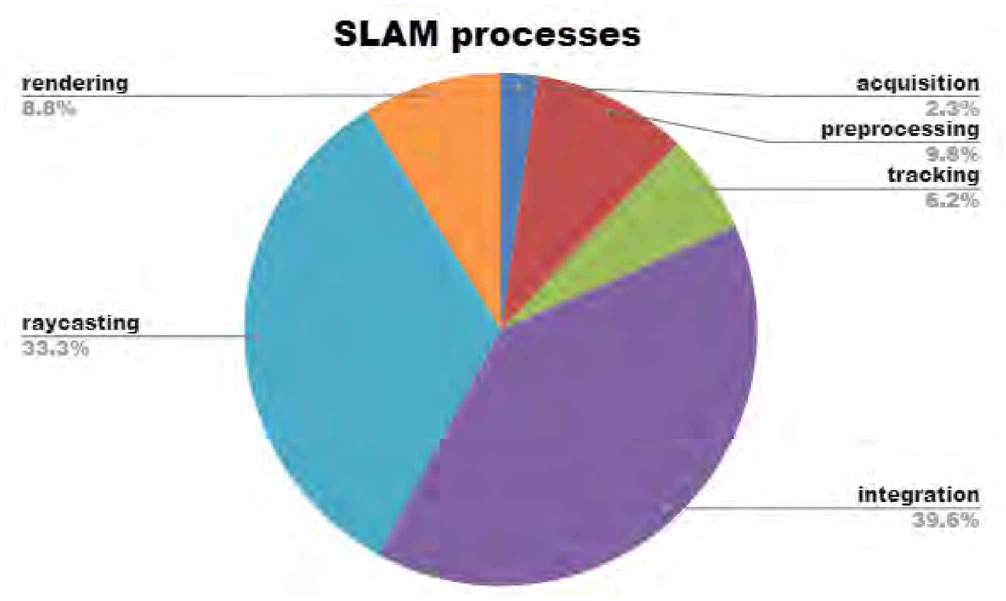


Figure 3.14: SLAM processes

Initially, the whole raycastkernel process was implemented with a hope to increase its performance.

a) **Interface**: An AXI4-Lite slave I/O protocol is combined with an AXI4 master I/O protocol to make an interface for the accelerator in order to be able to communicate.

b) **Memcpy**: The raycastkernel consists of three nested loops. Except that the third one initiates later on in the code. So in the third loop we need to load all the values for the interp operation. After the third loop ends and before the second one ends, we perform the grad function so we need to read them first. Once this and some other computations end, we generate the output and via memcpy two arrays are written back.

c) **Pipeline**: Due to the nature of algorithm and the code there are a lot of dependencies, as a result pipelining was not performed.

As we reached a stalemate with the previous implementation we decided to map to hardware only the raycast function of the code and not the whole raycastkernel. The Fig. 3.17 shows the stalemate as the initital and final implementation is the same.The raycast function possesses the heavy computational load of the raycasting step.

a) **Interface**: An AXI4-Lite slave I/O protocol together with an AXI4 master

I/O protocol will suffice as interface.

b) **Memcpy**: The raycast function consists of one loop. Before that we load 8 values for the interp operation and there is another one inside the loop where we need to load these 8 values per iteration. Furthermore, before returning from the accelerator we write back the output.

c) **Pipeline**: Firstly, we could not perform the pipeline pragma due to the nature of the original code. We had to rewrite a portion of the it, remove certain dependencies and make it able to be pipelined. The loop achieved II=8 due to the interp function that needs 8 loads.

Raycasting computes the implicit interface that corresponds to the current camera position estimation. For each surface pixel, the algorithm walks a single ray.The raycast function accesses the 3D space(256*256*256) by rays. The 3D space is 256^3 *4 bytes=64 MB.Each ray's direction, width and starting point are unique and depend on the frame view. Unfortunately, there is no spatial locality see Fig. 3.15, successive memory accesses have at least a 2D layer of the 3D space between them. One approach is to prefetch the data we need before entering the accelerator. For each ray the Arm processor computes the needed data for raycasting and saves them to a buffer which we give later to the raycast IP. This resulted in the raycasting step achieving 3.3285 seconds.This implementation underwent the same HLS optimizations as the previous one. However, this one achieved II=4. Instead of prefetching only data for one ray we decided to do a tiled implementation of the raycasting step. That means we prefetch a tile of 160*4 so as to increase the implementation's performance. Furthermore, by changing a little bit the interp function of the raycasting step of the algorithm we managed to achieve II=1. The HLS pragma pipeline was integrated in the raycasting step at the while loop in row 7 in algorithm 1. Unfortunately, the raycast step of the algorithm, even with the final implementation at 100 MHz, achieved a time execution of 1.9258s see Fig.3.16 and Fig. 3.17 .The design was not promising so it was not tested at higher frequencies.

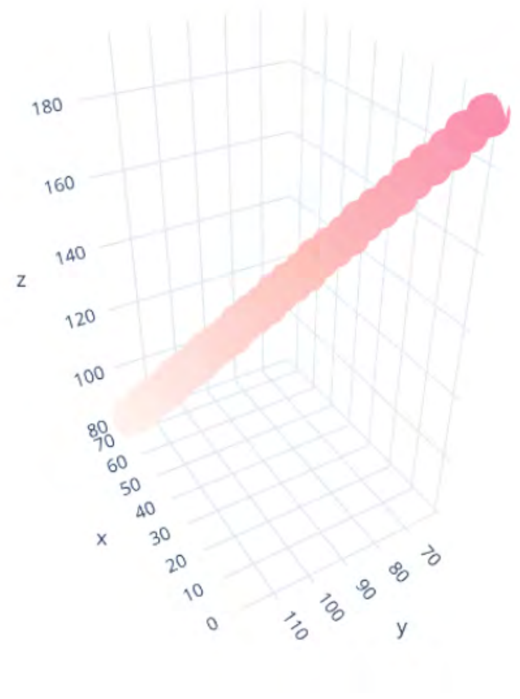


Figure 3.15: Access pattern of rays

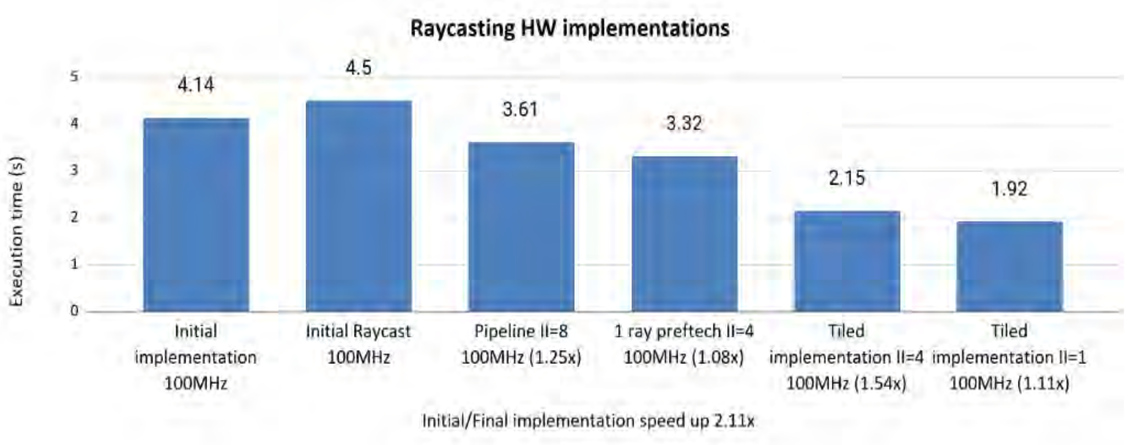


Figure 3.16: Raycasting HW implementations

We can see the area needed for both raycasting implementations in Table 3.2 and in Table 3.3 .

	BRAM_18K	DSP	FF	LUT	URAM
Mm2metersKernel	32(1%)	3($\sim$ 0%)	2935($\sim$ 0%)	3796(1%)	0
BilateralfilterKernel	34(1%)	17($\sim$ 0%)	4701($\sim$ 0%)	6537(1%)	0
IntegrateKernel	4($\sim$ 0%)	45(1%)	11660(2%)	14137(5%)	0
RaycastKernel	24(1%)	115(4%)	19114(3%)	31299(11%)	0
Raycast	18($\sim$ 0%)	69(2%)	16029(2%)	17743(6%)	0
Availabe Resources	1824	2520	548160	274080	0

Table 3.2: Ultrascale: Initial Kernels - Utilization

	BRAM_18K	DSP	FF	LUT	URAM
Mm2metersKernel	32(1%)	9(0%)	4476(0%)	4937(1%)	0
BilateralfilterKernel	45(2%)	216(8%)	166387(30%)	44873(16%)	0
IntegrateKernel	159(8%)	272(10%)	74097(13%)	51868(15%)	0
RaycastKernel	24(1%)	115(4%)	19114(3%)	31299(11%)	0
Raycast	496(27%)	160(6%)	29719(5%)	32888(11%)	0
Availabe Resources	1824	2520	548160	274080	0

Table 3.3: Ultrascale: Final Kernels - Utilization

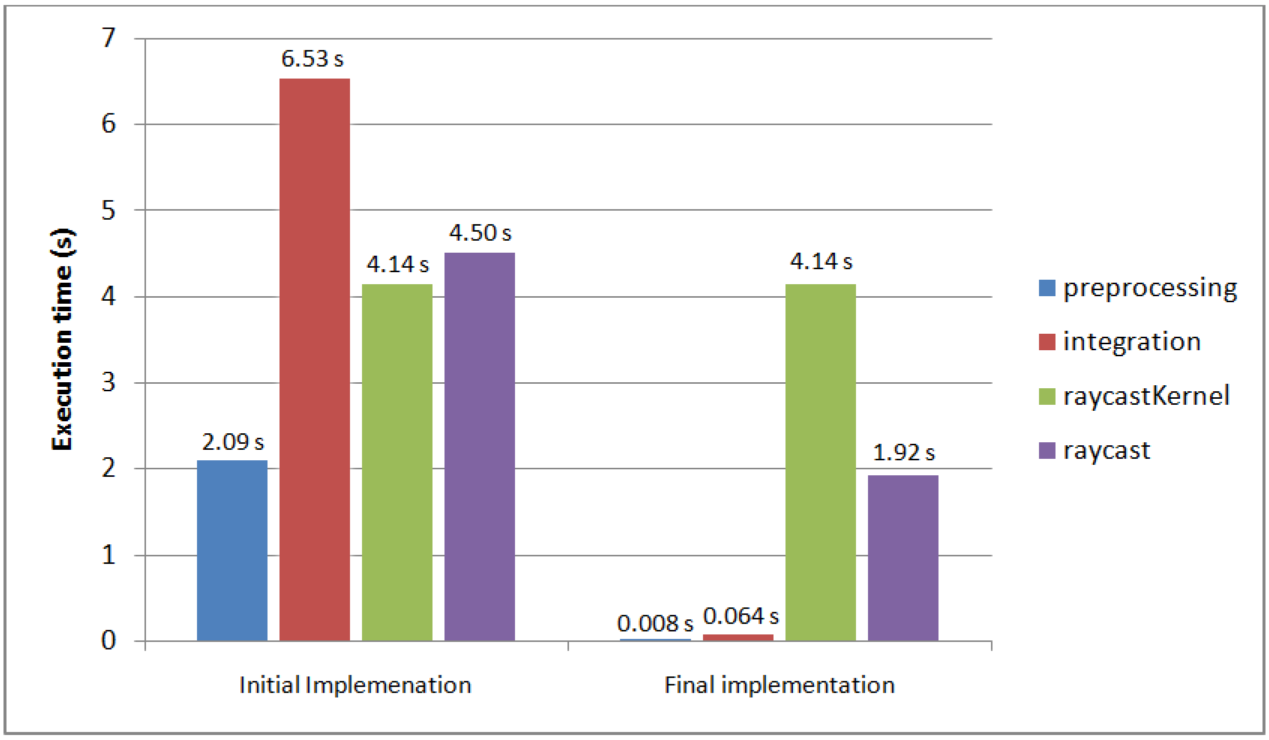


Figure 3.17: Initial and Final implementation of accelerators on Zynq Ultrascale+Mpsoc

3.7 Final Design overview

In our final design we implement only the bilateralfilterkernel and the integratekernel as the accelerator of the raycastkernel did not achieve better performance than the Arm of the Zynq® UltraScale+^{TM} MPSoC. The final design consists of four integratekernel accelerators and one bilaterfileterkernel accelerator with the optimizations we presented in sections earlier.After the generated RTL is exported from the Vivado HLS the accelerators are instantiated into the Vivado Design suite [28]. From Vivado HLS we can see the resource utilization for the bilateralfilterlkernel and integratekernel, as a result we can estimate the area of the whole project.However, the Post-Implementation report from the Vivado Design suite can give us a better picture of the utilization of the final design as shown in Fig. 3.18.

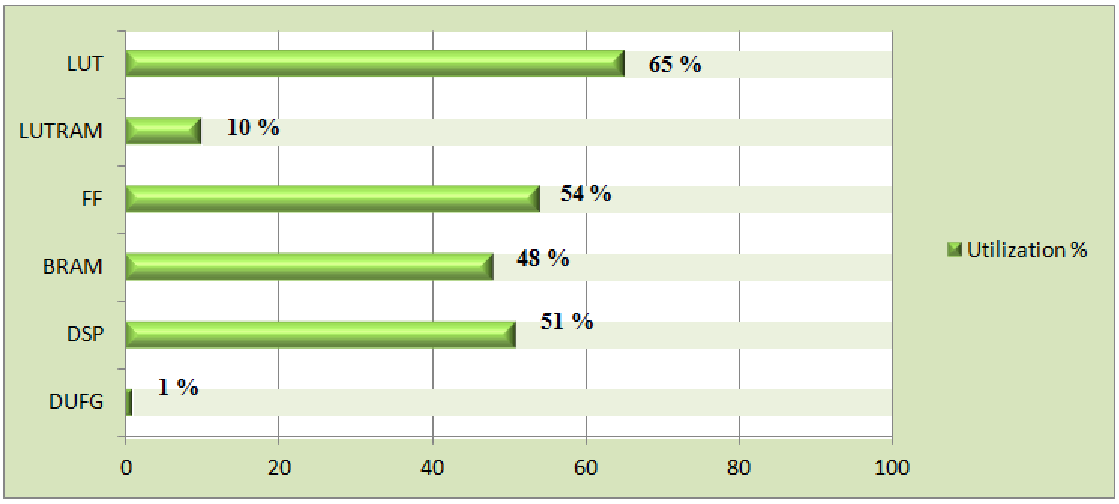


Figure 3.18: Post-Implementation utilization

We have to note that we decided not to implement another integratekernel or bilateralfilterkernel as accelerator in the design. We need some area available as potentially the raycastkernel could be implemented as an accelerator. With the help of the Petalinux tools [29] we accomplished to run the design on the ZCU102 kit.The Figure 3.19 shows the performance of the final design.

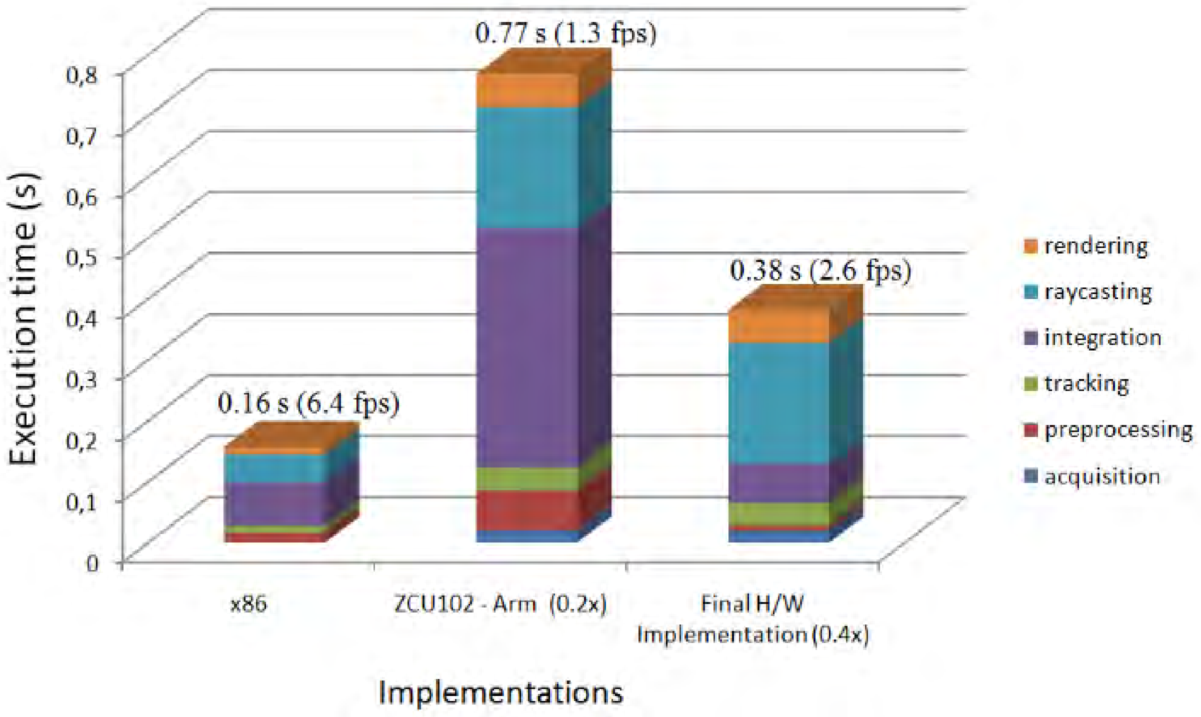


Figure 3.19: Final design performance and comparison

It is concluded that the Arm processor can give us a better performance than our initial implementation. However, with the optimization techniques we were able to achieve a better execution time of 0.38167s (2.63 FPS). Unfortunately, we were not able to match the performance of the x86.

3.8 Zedboard experimental results

The initial development of the KinectFusion implementation on a re-configurable platform was done on the Zedboard , but it turns out this FPGA is too small to accommodate such a large project. However, we have performance numbers for the Arm processor of the board in Fig. 3.20.

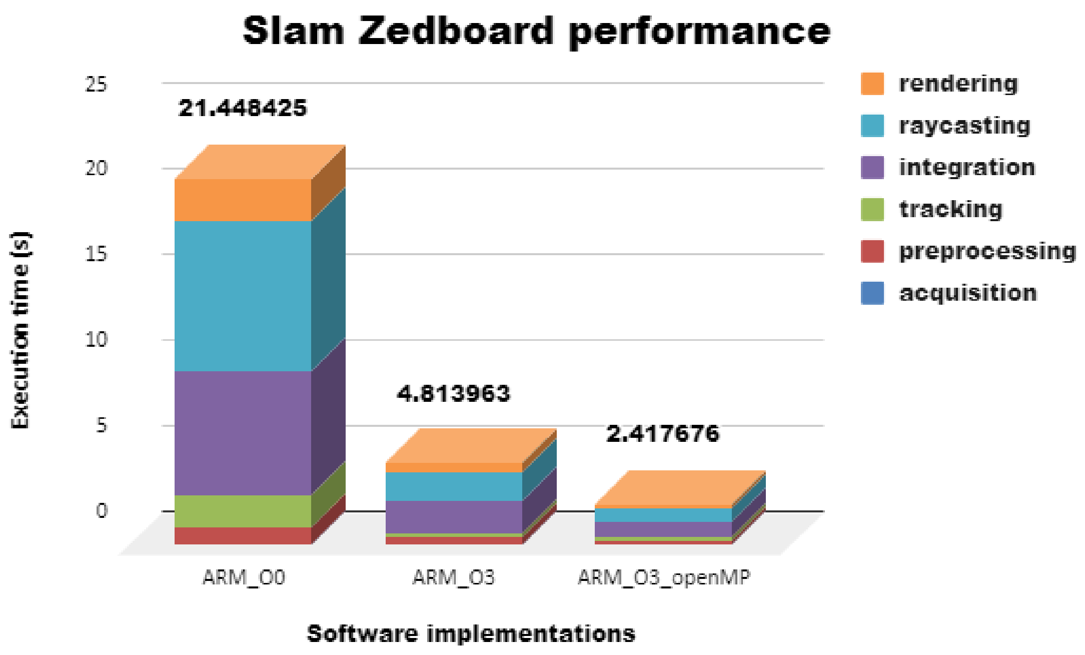


Figure 3.20: Execution time (s) - Implementations

3.8.1 Implementation of the Mm2 to Meters kernel on HLS

The development was the same as in the Section 3.3. However, the performance see Fig. 3.22 changes as well as the utilization, see Table 3.4 and Table 3.5, because of the different boards.

3.8.2 Implementation of the Bilateral Filter on HLS

The Zedboard does not offer us the area of Zynq Ultrascale+Mpsoc, as a result the improvement of the accelerator at an earlier stage compared to the Section 3.4. To be precise, the final implementation of the bilateralfiltekernel only gets to stage where II=13. Furthermore, the max clock frequency that is valid for the kernel is 125 MHz.

The final implementation of the preprocessing is with the same mindset as we discussed earlier Section 3.4.

The area demanded is in Table 3.4 and in Table 3.5 and computation time is show in Fig. 3.22 .

3.8.3 Implementation of the Integration step on HLS

Many of the approaches we discussed in Section 3.5 were initially tested on this FPGA. This section will provide also execution numbers for these implementations and justify our decision for not integrating them in Fig. 3.21.

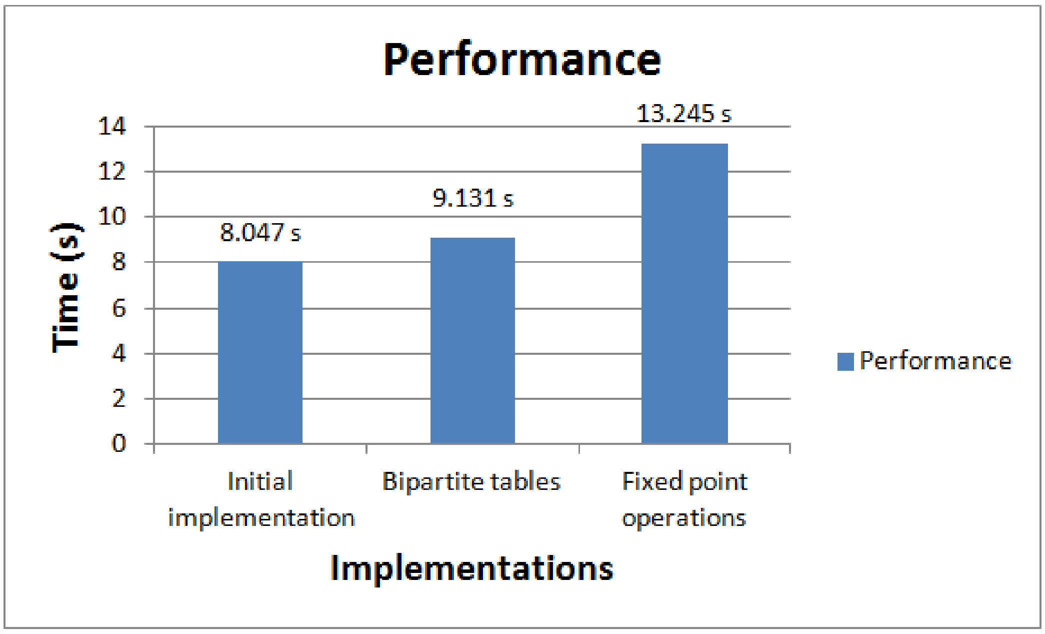


Figure 3.21: Integratekernel aborted Implementations

Unfortunately, on the Zedboard we were stuck on an earlier development stage due to area limitations. After the loop interchange the body of the loop achieved II=2. Furthermore, the loop was unrolled by a factor of two and the input volume array was partitioned by a factor of two. The max valid fabric clock frequency that the design was executed is 125 MHz.

The resources given are in Table 3.4 and in Table 3.5 and the performance is shown in Fig. 3.22.

3.8.4 Implementation of the Raycasting step on HLS

On the Zedboard the whole raycastkernel process was implemented as in Section 3.6. We reached a stalemate with this implementation ,as we discussed earlier, and we decided to map to hardware only the raycast function. Unfortunately,most of the raycasting step was developed on the ZCU102 kit.The only implementation tested was the one where we pipelined the raycast function to the point of II=8.As a result, no other execution numbers are available, the only ones are in Fig. 3.22. The area is shown in Table 3.4 and in Table 3.5.

	BRAM_18K	DSP	FF	LUT
Mm2metersKernel	32(11%)	3(1%)	3556(3%)	4260(8%)
BilateralfilterKernel	34(12%)	17(7%)	5282(4%)	7618(14%)
IntegrateKernel	22(7%)	48(21%)	13214(12%)	19364(36%)
RaycastKernel	32(11%)	143(65%)	24870(23%)	39714(74%)
Available resources	280	220	106400	53200

Table 3.4: Zedboard: Initial Kernels - Utilization

	BRAM_18K	DSP	FF	LUT
Mm2metersKernel	32(11%)	9(4%)	5242(4%)	5491(10%)
BilateralfilterKernel	39(13%)	56(25%)	34166(32%)	20570(38%)
IntegrateKernel	22(7%)	142(64%)	31455(29%)	45156(84%)
RaycastKernel	32(11%)	143(65%)	24870(23%)	39714(74%)
Available resources	280	220	106400	53200

Table 3.5: Zedboard: Final Kernels - Utilization

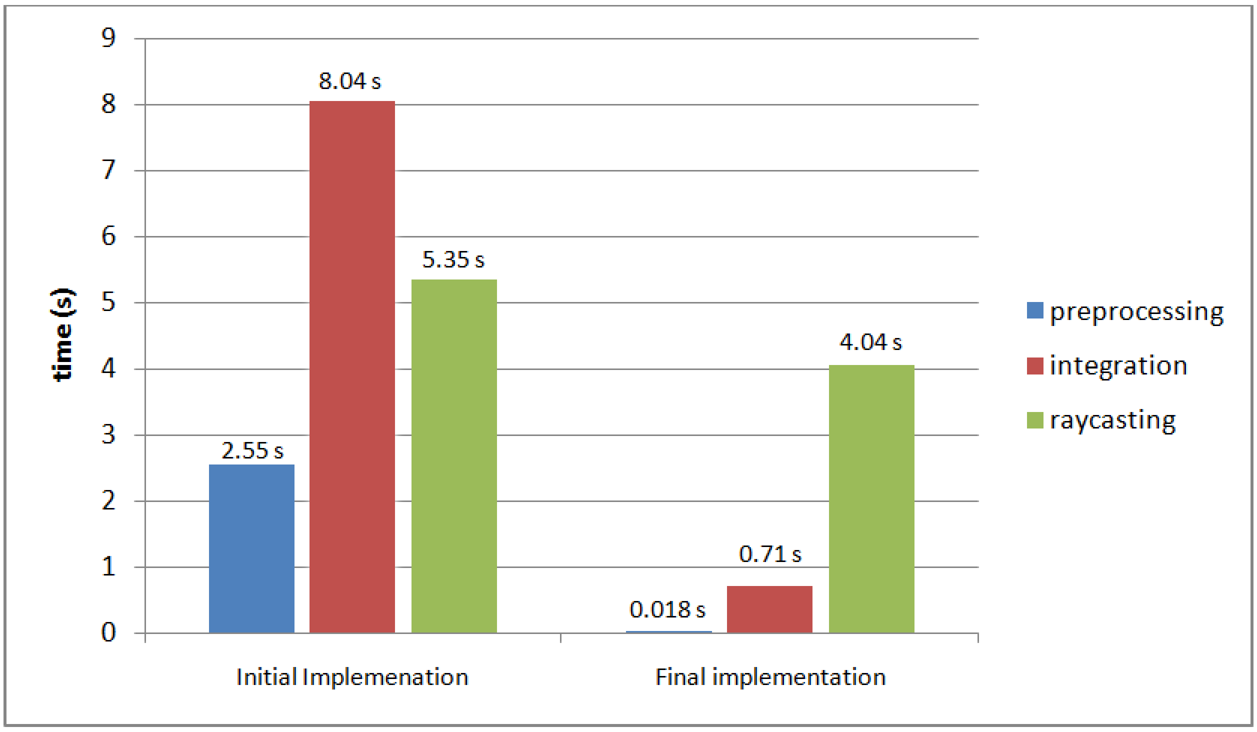


Figure 3.22: Initial and Final implementation of accelerators on Zedboard

3.8.5 Zedboard HW implementations

As the size of the Zedboard is not big enough to accommodate integratekernel and bilateralfilterkernel together in an implementation we have to choose one of them to implement each time. The raycasting step in HW did not achieve a better execution time than the SW implementation so no approach with a raycastekernel accelerator will be presented in Fig. 3.23.

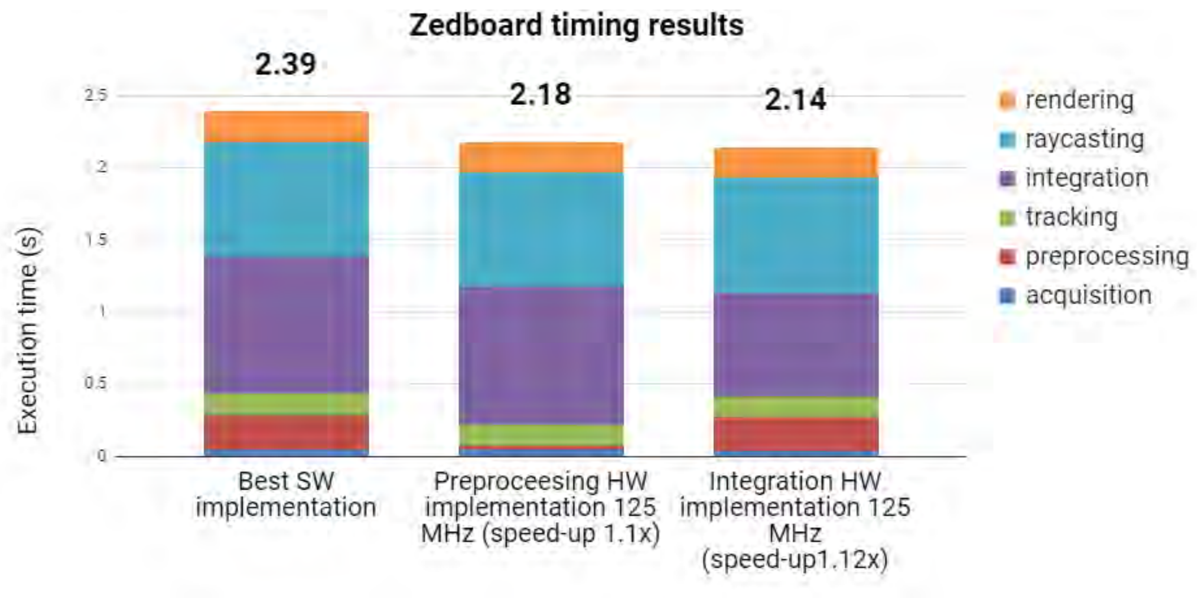


Figure 3.23: Hardware implementation of accelerators on Zedboard

Chapter 4

Conclusion

In conclusion, the execution time of the KinectFusion algorithm on a re-configurable platform shows promising results. Optimizations both on algorithmic and architectural optimizations were tested. The HLS design, despite lacking the precision of a RTL design, can still be used with satisfying results that are comparable to the x86's . An implementation of nearly 3 FPS is far behind any real time application. However, there is still room for future improvements in order to increase the performance of the design.

4.1 Future work

In this thesis a HLS based design is implemented and presented on the ZCU102, however a RTL design can be examined. With a design so much closer the hardware of the re-configurable platform , there is a good chance that an extremely good performance will be achieved. However, there are still plenty of ideas that we could try to apply to the HLS based implementation.

Let's make it clear that not all kernels can fit together on a Zynq-7020 FPGA, however on a Zynq©UltraScale+MPSoC that may be possible. So in the future, a design with all kernels integrated can be researched. However,as always we choose the right candidates first for an accelerator. By resource usages alone raycastkernel should be an accelerator, but in our implementation we could not design an IP good enough to achieve any good performance numbers. More research is needed so as to make an approach of the raycasting step that will be good enough to be integrated as an accelerator . In this implementation, the m-axi and s-axiilite interface used in all kernels can be altered. An interface with an AXI-streaming protocol may give better results. Furthermore, it enables the usage of a Zynq High-Performance port together with the AXI Direct Memory Access (DMA). A DMA may relieve some stress from the processor as it was used for memory transfers.

Bibliography

[1] Cesar Cadena et al. “Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age”. In: *IEEE Transactions on Robotics,* vol. 32, no. 6, (2016), pp. 1039–1332. DOI: 10.1109/TRO.2016.2624754.

[2] M. Abouzahir et al. “Embedding SLAM algorithms: Has it come of age?” In: (2018). DOI: 10.1016/j.robot.2017.10.019.

[3] *Programmable-logic and application-specific integrated circuits.* URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[4] I. Kuon and J. Rose. “Measuring the gap between FPGAs and ASICs”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 26, no. 2, (2007), pp. 203–215. DOI: 10.1109/TCAD.2006.884574.

[5] *System on chip(Soc).* URL: https://anysilicon.com/what-is-a-system-on-chip-soc/.

[6] *How Microsoft Kinect Sensor Works.* URL: https://de.mathworks.com/help/hdlcoder/examples/getting-started-with-hardware-software-codesign-workflow-for-zynq-ultrascale-mpsoc-devices.html.

[7] Yan Pei et al. “SLAMBooster: An Application-aware Online Controller for Approximation in Dense SLAM”. In: (2018). URL: https://arxiv.org/abs/1811.01516.

[8] L. Nardi et al. “Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM”. In: *IEEE International Conference on Robotics and Automation (ICRA),* (2015), pp. 5783–5790. DOI: 10.1109/ICRA.2015.7140009.

[9] Richard A. Newcombe et al. "KinectFusion: Real-Time Dense Surface Mapping and Tracking". In: *10th IEEE International Symposium on Mixed and Augmented Reality,* (2011). DOI: 10.1109/ISMAR.2011.6092378.

[10] *SLAMBench, an open source tool.* URL: https://github.com/pamela-project/slambench.

[11] *Digilent ZedBoard Zynq®-7000 ARM/FPGA SoC Development Board.* URL: https://www.xilinx.com/products/boards-and-kits/1-elhabt.html.html.

[12] *Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit.* URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[13] *ZCU102 Evaluation Board User Guide.* URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf.

[14] J. Piat D. Törtei Tertei and M. Devy. ""FPGA design of EKF block accelerator for 3D visual SLAM". In: *Computers and Electrical Engineering* vol. 55,no. 2, (2016), pp. 1339–1351. DOI: 10.1016/j.compeleceng.2016.05.003.

[15] Quentin Gautier, Alric Althoff, and Ryan Kastner. "FPGA Architectures for Real-time Dense SLAM". In: *IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP),* (2019). DOI: 10.1109/ASAP.2019.00-25.

[16] Samunda Perera et al. "Semi-dense visual odometry for a monocular camera". In: *Proceedings of the IEEE International Conference on Computer Vision,* (2013). DOI: 10.1109/WACV.2015.144.

[17] *Iterative closest point algorithm.* URL: https://en.wikipedia.org/wiki/Iterative_closest_point.

[18] J. Engel, J. Sturm, and D. Cremers. "Motion Segmentation of Truncated Signed Distance Function Based Volumetric Surfaces". In: *2015 IEEE Winter Conference on Applications of Computer Vision,* (2015), pp. 1449–1456. DOI: 10.1109/ICCV.2013.183.

[19] R.Nane et al. “A Survey and Evaluation of FPGA High-Level Synthesis Tools”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 35, no. 10, (2016), pp. 1591–1604. DOI: 10.1109/TCAD.2015.2513673.

[20] *Vivado HLS User Guide 2018.* URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.

[21] *Memcpy in C/C++.* URL: https://www.geeksforgeeks.org/memcpy-in-cc/.

[22] *Vivado HLS Optimazation Methodology Guide 2017.* URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug1270-vivado-hls-opt-methodology-guide.pdf.

[23] *Clamping (graphics).* URL: https://en.wikipedia.org/wiki/Clamping_(graphics).

[24] *Using Faster Exponential Approximation.* 2014. URL: https://codingforspeed.com/using-faster-exponential-approximation.

[25] P. Štukjunger M. Bečvář. “Fixed-Point Arithmetic in FPGA”. In: vol. 45, no. 2, 2005, pp. 67–72.

[26] *Reduce Power and Cost by Converting from Floating Point to Fixed Point.* URL: https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf.

[27] Michael J. Schulte and James E. Stine. *Symmetric Bipartite Tables for Accurate Function Approximation.*

[28] *Vivado Design Suite Tutorial 2018.* URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug940-vivado-tutorial-embedded-design.pdf.

[29] *Petalinux Tools reference guide 2018.* URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-petalinux-tools-reference-guide.pdf.

[30] *Xilinx Zynq UltraScale+ MPSoC*. URL: https://de.mathworks.com/help/hdlcoder/examples/getting-started-with-hardware-software-codesign-workflow-for-zynq-ultrascale-mpsoc-devices.html.

[31] he Yu and Chen Shengyong. "Advances in sensing and processing methods for three-dimensional robot vision". In: *International Journal of Advanced Robotic Systems,* vol. 15, (2018). DOI: 10.1177/1729881418760623.

[32] Nag Arunava and Deshmukh Sanket. *Real Time Tracking System using 3D Vision*. 2015. DOI: 10.13140/RG.2.1.3513.4489.