Volos 2020

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Implementation of the Tracking Algorithm of a Visual SLAM System on FPGAs

Diploma Thesis

Biliouri Viktoria

Supervisor: Bellas Nikolaos

2nd Committee Member: Lalis Spyros

3rd Committee Member: Katsaros Dimitrios

Βόλος 2020

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Υλοποίηση του Αλγορίθμου Tracking ενός Visual SLAM Συστήματος σε FPGAs

Διπλωματική Εργασία

Μπιλιούρη Βικτώρια

Επιβλέπων: Μπέλλας Νικόλαος

2ο Μέλος Επιτροπής: Λάλης Σπύρος

3ο Μέλος Επιτροπής: Κατσαρός Δημήτριος

Dedicated to my Family…

Volos 2020

Acknowledgements

For and foremost, I would like to thank my supervisor to this Diploma Thesis, Mr Bellas Nikolaos, for his support and guidance, not only for the completion of this Thesis, but also for the implementation of other projects during my studies. His contribution was decisive for the completion of my studies, in the Department of Electrical and Computer Engineering of University of Thessaly. I would also like to express my deep appreciation to Doctorate Candidate Gkeka Maria Rafaela, for her help and useful consultancy during the Implementation of my Diploma Thesis, and I wish her to fulfil all her dreams.

Additionally, I could not miss the chance to express my gratitude to my family for the support that they are showing, to every step of my life. Without you I would have accomplished nothing.

Last but not least, I feel blessed of having my friends, who, by supporting and understanding me during the past five years, they have filled me with wonderful memories, which I will be reminiscing for the rest of my life.

Βόλος 2020

Ευχαριστίες

Θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες, στον Επιβλέποντα Καθηγητή της παρούσας Διπλωματικής Εργασίας, κ. Μπέλλα Νικόλαο, για την αμέριστη στήριξη και καθοδήγηση του τόσο σε αυτήν, όσο και σε άλλες εργασίες και μαθήματα κατά τη διάρκεια της ακαδημαϊκής μου πορείας. Η συνεισφορά του, κατέστη εξαιρετικά χρήσιμη στην ολοκλήρωση των Σπουδών μου, στο Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, του Πανεπιστημίου Θεσσαλίας. Επιπλέον, θα ήθελα να ευχαριστήσω την Υποψήφια Διδάκτορ Γκέκα Μαρία Ραφαέλα, για την πολύτιμη βοήθεια και τις χρήσιμες συμβουλές της, κατα τη διάρκεια της υλοποίησης της Διπλωματικής μου Εργασίας, και να της ευχηθώ καλή σταδιοδρομία.

Ακόμα, δεν θα μπορούσα να παραλείψω να εκφράσω την ευγνωμοσύνη μου στην οικογένεια μου, για την υποστήριξη που δείχνουν σε κάθε βήμα της ζωής μου, χωρίς την οποία θα ήταν εξαιρετικά δύσκολο να επιτύχω.

Τέλος, νιώθω την ανάγκη να ευχαριστήσω από καρδιάς τους φίλους μου, που με την κατανόηση και την στήριξη τους κατα τη διάρκεια αυτών των πέντε ετών, μου πρόσφεραν αναμνήσεις που θα θυμάμαι για όλη μου τη ζωή.

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

«Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/ και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής».

Ο/Η Δηλών/ούσα

(Υπογραφή)
Μπιλιούρη Βικτώρια
Ημερομηνία

ABSTRACT

Today, in the era of technological advance, the use of Visual SLAM Systems (Simultaneous Localisation and Mapping Systems) has become a common idea. The Visual SLAM Systems form a 3D view of an area, by renewing the formed map in a repeatable manner, in order to to track and map all of the objects into it. This kind of systems are today utilized in various domains such as robotics, virtual and augmented reality, and they are chosen by the majority of the most important enterprises in the world of Computer Sciences. A well known SLAMbench algorithm is the KinectFusion algorithm, which processes via a number of stages, input data, deriving from depth frames, and reforms a 3D view of an area by precisely placing the objects into it. The FPGA (Field Programmable Gate Array) platforms, the use of which increases day by day, are reprogrammable structures, that contain a great number of logic gates, clocks, generators etc. and are used as complete circuits, for the execution of specific tasks. The purpose of this Diploma Thesis is the hardware implementation of the tracking algorithm of KinectFusion process, its execution on FPGAs and the optimization of its latency. The laboratory section of this Thesis, was implemented using Vitis Unified Software Platform and Vivado HLS Xilinx tools, on evaluation board Zynq® UltraScale+™ MPSoc, with input the ICL-NUIM Dataset. The optimum execution time achieved in this application for the track kernel, is 0.00146 sec per frame.

ΠΕΡΙΛΗΨΗ

Στον σημερινό κόσμο της καλπάζουσας τεχνολογίας, γίνεται συχνή αναφορά στα Visual SLAM συστήματα , δηλαδή στα συστήματα Ταυτόχρονου Εντοπισμού και Χαρτογράφησης, τα οποία με χρήση Αλγορίθμων SLAM, σχηματίζουν απεικόνιση τριών διαστάσεων ενός χώρου, ανανεώνοντας με επαναλαμβανόμενο τρόπο τον σχηματισμένο χάρτη, με αποτέλεσμα την εύρεση και τοποθέτηση όλων των αντικειμένων σε αυτόν. Τέτοια συστήματα χρησιμοποιούνται σήμερα σε εφαρμογές ρομποτικής, εικονικής πραγματικότητας κ.α., και επιλέγονται από την πλειονότητα των μεγάλων εταιριών του κόσμου τον Υπολογιστών. Ένας πολύ διαδεδομένος αλγόριθμος SLAM είναι και ο αλγόριθμος Kinect Fusion, ο οποίος λαμβάνοντας ως είσοδο εικόνες με ενδείξεις βάθους των αντικειμένων ενός χώρου, επεξεργάζεται τα δεδομένα μέσα από μια διαδικασία αρκετών σταδίων και ανασχηματίζει την τρισδιάστατη απεικόνιση του χώρου, με ακριβή τοποθέτηση όλων των αντικειμένων σε αυτόν. Οι πλακέτες FPGA (Field Programmable Gate Array ή συστοιχία επιτόπια προγραμματιζόμενων πυλών) , η χρήση των όποιων όλοενα και αυξάνεται τα τελευταία χρόνια, είναι επαναπρογραμματιζούμενες δομές που διαθέτουν μεγάλο αριθμό λογικών πυλών, ρολογιών, γεννητριών κ.α και χρησιμοποιούνται ως ολοκληρωμένα κυκλώματα για την εκτέλεση συγκεκριμένων λειτουργιών. Αντικείμενο αυτής της Διπλωματικής Εργασίας, είναι η hardware υλοποίηση του αλγορίθμου του σταδίου tracking (στάδιο του εντοπισμού αντικειμένων) του αλγορίθμου KinectFusion, και η εκτέλεση του σε FPGAs, καθώς και η βελτιστοποίηση του αλγορίθμου με στόχο την ελαχιστοποίηση του χρόνου εκτέλεσης και της καθυστέρησης του. Το εργαστηριακο μέρος της εργασίας, υλοποιήθηκε με τη χρήση του εργαλείου Vitis Unified Software Platform και του Vivado HLS της Xilinx, στην πλακετα Xilinx Zynq® UltraScale+™ MPSoc ZCU102, με είσοδο το ICL-NUIM Dataset. Ο βέλτιστος χρόνος του αλγορίθμου tracking που επετεύχθει είναι τα 0.0014 sec/ frame.

Table of Contents

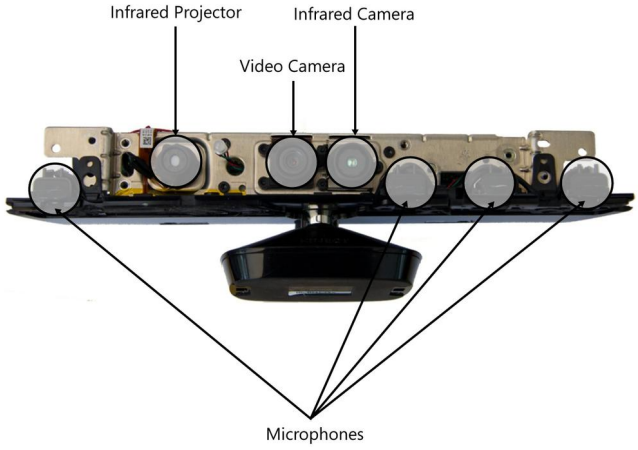
Chapter 1

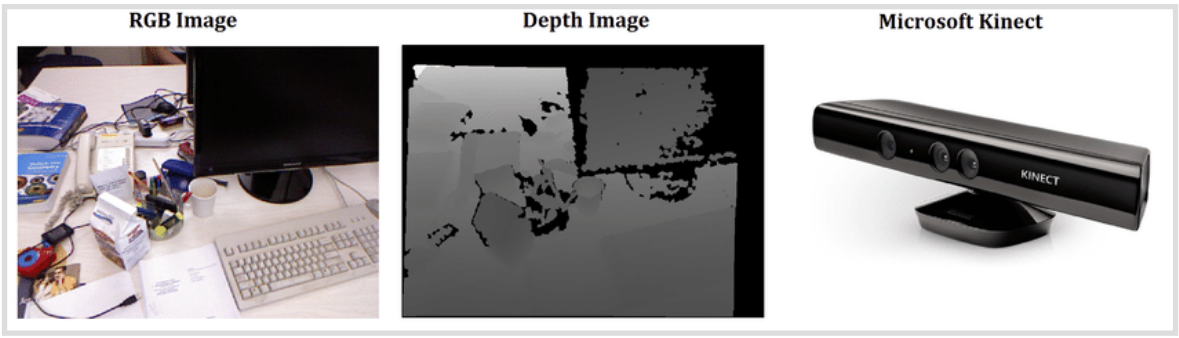
Introduction

1.1 General Overview

Simultaneous Localisation and Mapping, (SLAM) is a computational technique, which aims to reconstruct a 3D display of a specific area, and form a map by refreshing it and adding the exact position of the objects that exist in it. The problem that SLAM Systems deal with, exists in the world of Computer Sciences for a large period of time, though unresolved. However, today, the creation of several SLAM algorithms, came to offer a solution, and rendered the SLAM process, capable to be used by computer machines.

One of the most popular SLAM algorithms, is KinectFusion, an algorithm originally developed my Microsoft, and designed to cooperatively function with the camera of Kinect Motion Controller, a device that combines input sensors and depth-sensing cameras, and which is used for gaming (Kinect Xbox) and non-gaming applications in robotics, medicine, health care etc.



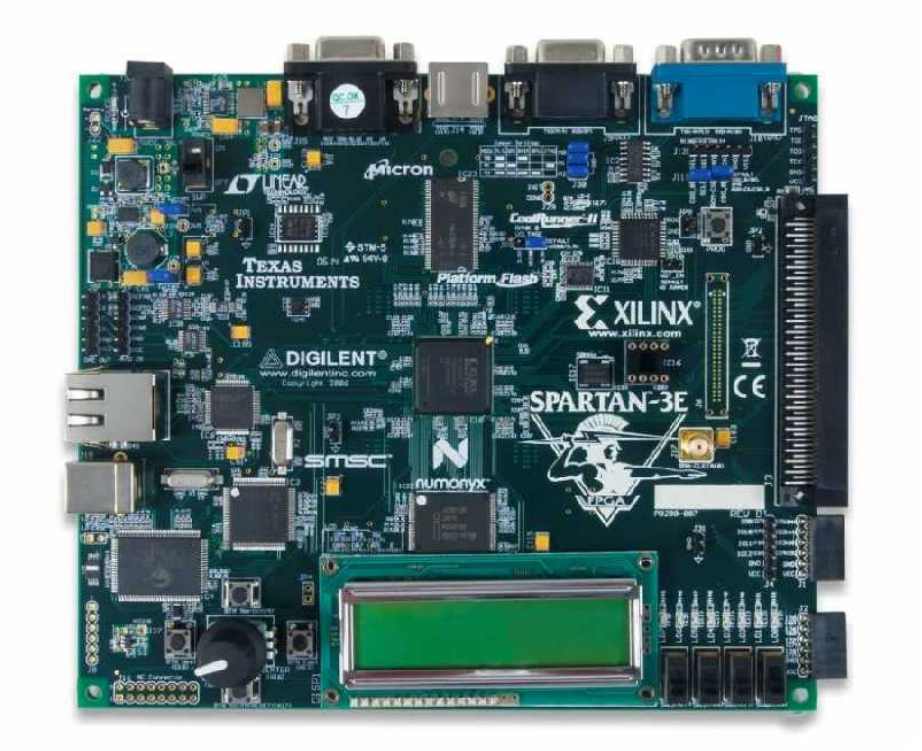


1.1. Kinect Motion Controller

Thus, for the purpose of this project, we have used the ICL-NUIM Dataset of RGB frames which is further analysed below. The process of KinectFusion Algorithm, composes of several stages and kernels, each one of them, is an integral part of the maintenance of the algorithm's accuracy. The responsibility of the Track kernel, is to track the exact position of the objects of an indoor scene, by comparing their current position with the 3D pose captured in the previous iteration of the algorithm. Due to the demanding computational process and the continuous data transfer of the algorithm of the kernel, apart from the software implementation, in terms of this Thesis, the track kernel was implemented in hardware, in order to function, using the FPGA platform included in Zynq® UltraScale+™ MPSoC ZCU102.

1.2 FPGA

FPGA (Field-Programmable Gate Array) [1] is an integrated circuit designed to be programmed by the customer, after its manufacturing process, and it is currently being used for numerous research and development problems, due to its reprogrammability. This feature, diversify its use, from the ASIC (Application-Specific Integrated Circuit) chips, as their functioning is pre-defined by the manufacturer, and the customer can only use them for specific applications.



1.2. Xilinx Spartan 3E FPGA

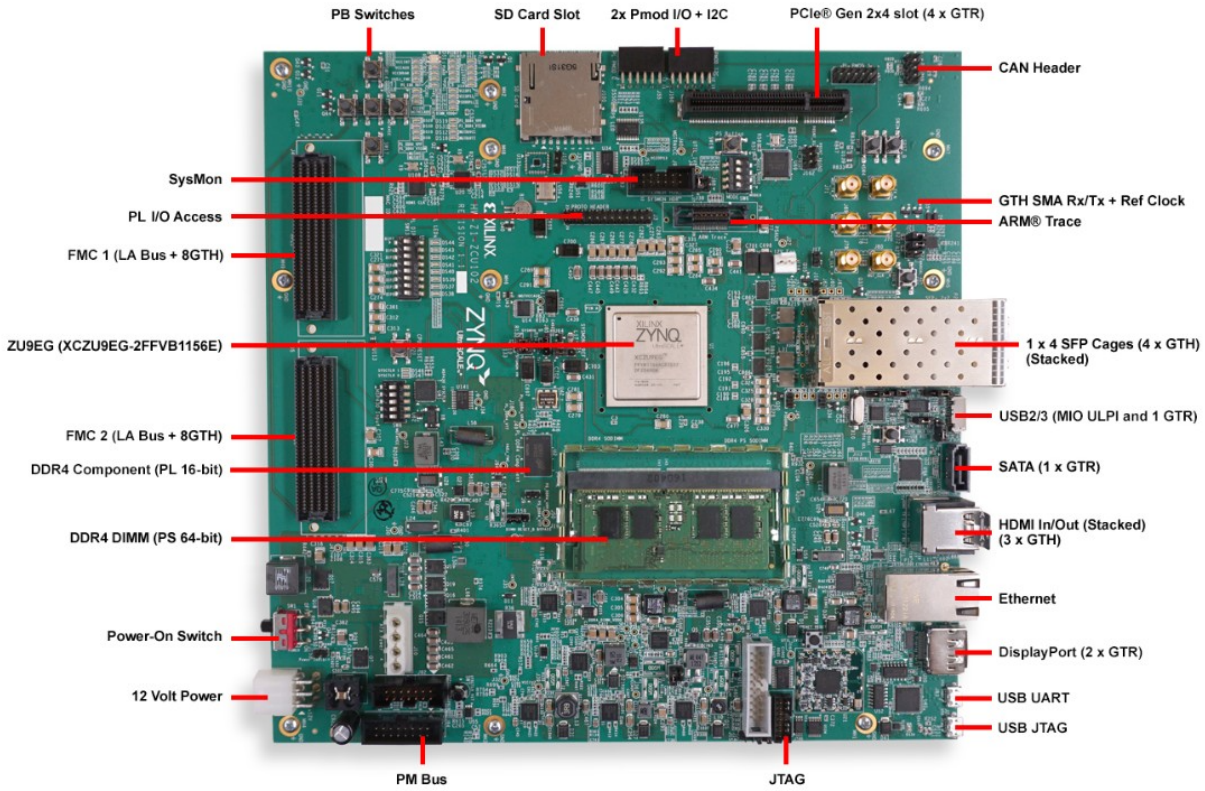
The FPGA platforms consist of an array of programmable logic blocks which are wired together by interconnects, and can be reconfigured to execute different complex combinational functions. Also, they contain memory elements, such as Flip Flops, or more complicated memory structures, such as Block RAMs, ROMs etc. Furthermore, it is common that FPGAs are used in embedded systems like System-On-Chips (Soc) because of the ability of simultaneous execution of software (SW) and hardware (HW). For this reason, for the execution of KinectFusion Algorithm, we have chosen the evaluation board Zynq Ultrascale+ MpSoc ZCU102.

1.2.1 Xilinx Zynq® UltraScale+™ MPSoc ZCU102

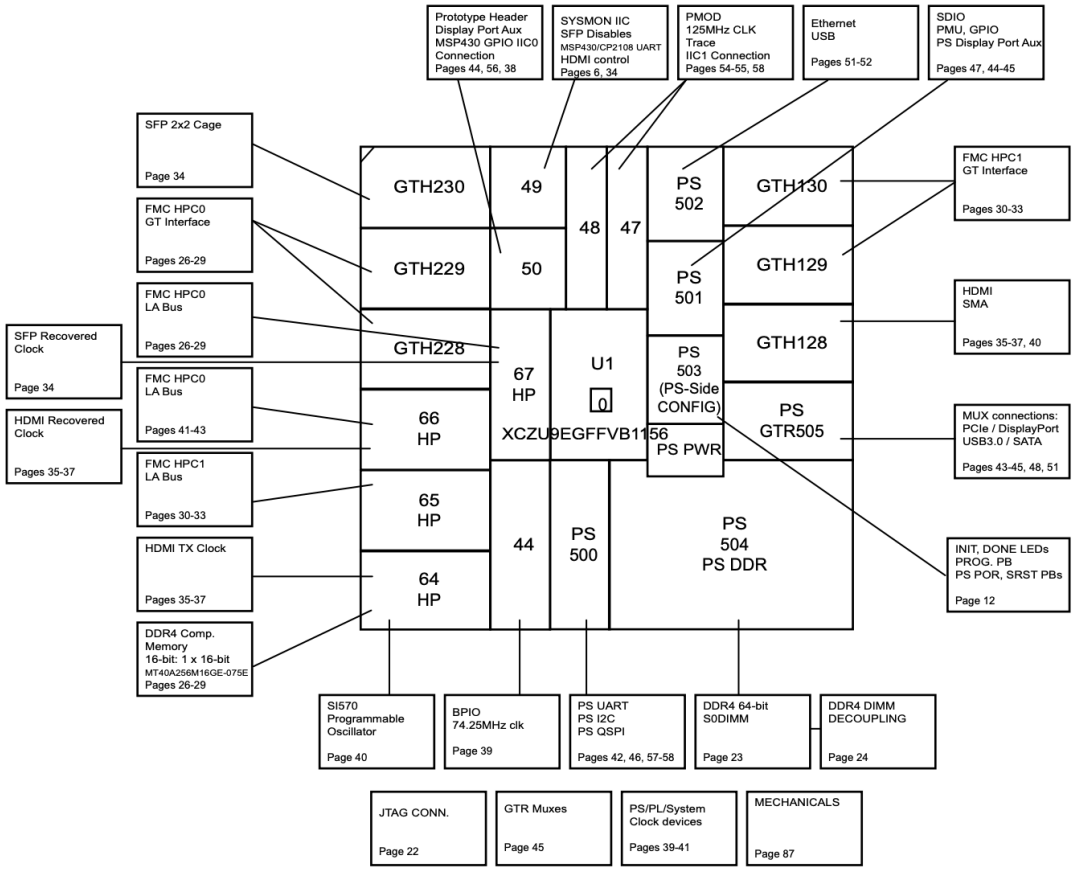
The Xilinx Zynq® UltraScale+™ MPSoc ZCU102 Evaluation Kit [2] , is a set of devices that is frequently used in the design of Automotive, Industrial, Video and Communications applications. The MPSoC (Multiprocessor System-on-Chip) ZCU102 platform, which combines a powerful processing system and user-programmable logic, features a Zynq UltraScale+ MPSoC device with a quad-core ARM® Cortex-A53, dual-core Cortex-R5 real-time processors, and a Mali-400 MP2 graphics processing unit based on Xilinx's 16nm FinFET+ programmable logic fabric. The Kit's ZCU102 Board supports all major peripherals and interfaces enabling development for a wide range of applications.

The ZCU102 is a general purpose evaluation board for rapid-prototyping based on the Zynq® UltraScale+™ XCZU9EG-2FFVB1156E MPSoC (Multiprocessor System-on-Chip). High speed DDR4 SODIMM and other memory interfaces, FMC expansion ports, multi-gigabit per second, serial transceivers, various peripheral interfaces and FPGA logic for user customized designs, render the ZCU102 evaluation board, a flexible reprogrammable platform.

The ZCU102 board, is designed to be easily programmed using Xilinx development platforms. For this reason, this project is implemented in Xilinx Vitis Unified Software Platform for the Software implementation of KinectFusion algorithm and Xilinx Vivado HLS for the Hardware implementation of the Track kernel.



1.3. Xilinx Zynq® UltraScale+™ MPSoc ZCU102



1.4. ZCU102 Block diagram

1.3 Application Design Tools

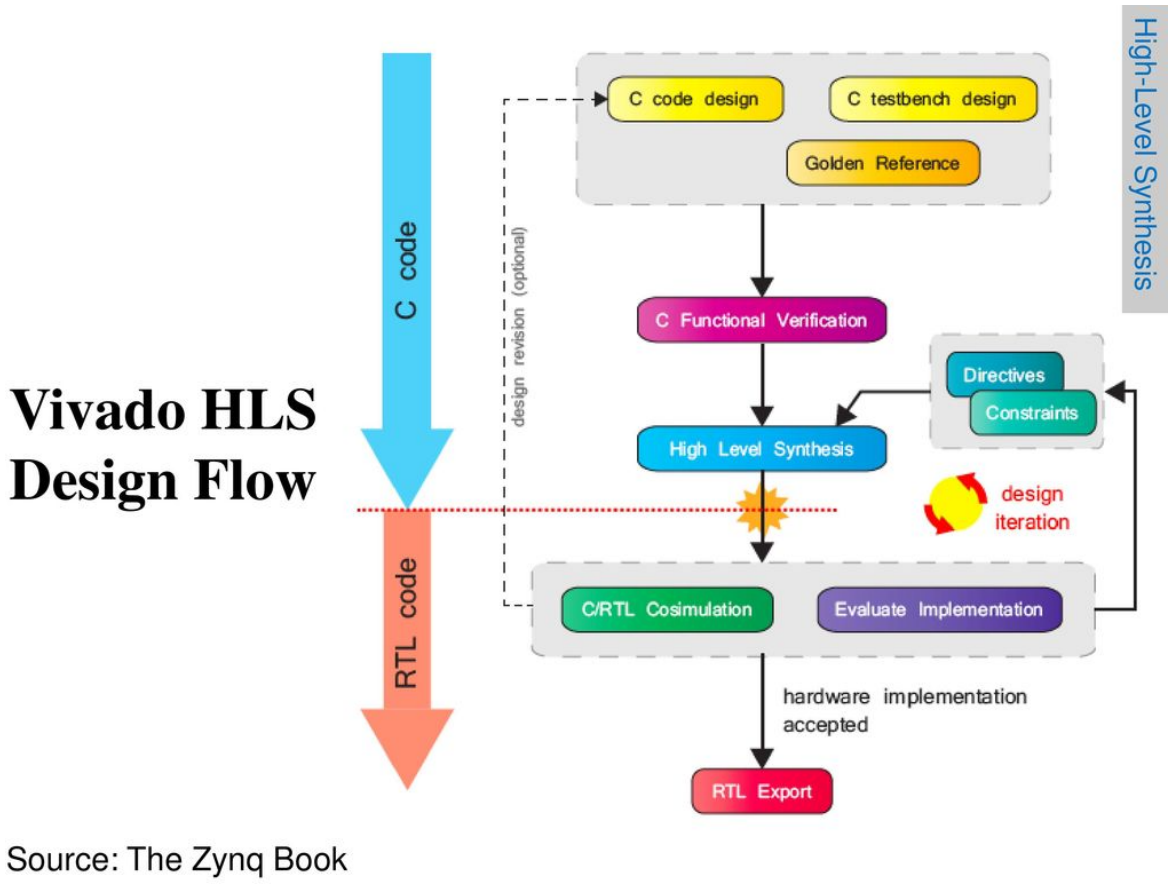
1.3.1 Vitis Unified Software Platform

Vitis [3] is a new Software design tool, that allows the development of software applications using an API, such as the OpenCL API to execute Hardware kernels designed by using the Vivado Design Suite. The Vitis core development kit [4] supports running the Software application on an embedded processor platform running Linux, such as the Zynq® UltraScale+™ MPSoc ZCU102, that we have used for our implementation. For the Software development of each program Vitis uses the OpenCL API to provide connection and simultaneous execution of the Hardware kernels and the data transaction among them. In our project, for the Hardware development of the Track kernel, we have used the Vivado HLS tool.

1.5. Vitis Software Development Kit Content

1.3.2 Vivado HLS

Vivado HLS (High-Level Synthesis) [5] tool , is a hardware accelerator, which is used for the design and execution of ips using C/C++/System C programming language, and it provides synthesis and simulation results, in order to control the accuracy and timing of each kernel. Vivado HLS is offered by Xilinx Vivado Design Suite and it facilitates the implementation and connectivity of each ip, as the programming language is automatically converted to RTL without any need of using HDL (Hardware Design Languages) (as VHDL, Verilog). The statistical and Analysis results provided by the end of the Synthesis process, give programmer the opportunity to check for possible mistakes, practice Timing and Spacing Optimizations, and in general observe step by step the functioning of their program.



1.6. Vivado HLS Design Flow Diagram

1.3.3 OpenCL API

Open Computing Language (OpenCL) [6] is a framework for parallel programming of heterogenous systems that include Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), Field-Programmable Gate Arrays (FPGAs) and other processors and hardware accelerators. OpenCL [7] views a computing system as consisting of a number of compute devices, it defines a C-language for writing programs, and it typically consists of multiple processing elements, called compute units. Each function executed on an OpenCL device is called kernel. In addition to its C-like programming language, OpenCL defines an application programming interface (API) that allows programs running on the host to launch kernels on the compute devices and manage device memory, which is (at least conceptually) separate from host memory. The OpenCL standard [8] defines host APIs for C and C++; third-party APIs exist for other programming languages and platforms such as Python, Java, Perl and .NET. An implementation of the OpenCL standard consists of a library that implements the API for C and C++, and an OpenCL C compiler for the compute device(s) targeted.

Chapter 2

2.1 SLAM

Simultaneous Localization and Mapping (SLAM) [9] is, as mentioned above, the computational problem of generating a 3D reconstruction of an indoor scene, by tracking and placing the objects in a 3D map. Though complicated this process may sound, there is a variety of algorithms that are used efficiently in SLAM Systems. SLAM algorithms [10] are tailored to the available resources, hence not aimed at perfection but at operational compliance. Several applications that utilize the SLAM technique are self-driving cars, unmanned aerial vehicles, autonomous underwater vehicles, domestic robots etc.

SLAM algorithms [11] can be categorised by the complexity of the calculations and the quality of each algorithm, as Sparse SLAM and Dense SLAM. Sparse SLAM algorithms are not as complicated as Dense SLAM algorithms, but the former do not provide as accurate results as the latter. Semi Dense SLAM are created to fill the gap of the other two types.

Also, the SLAM algorithms are characterised as Direct and Indirect, depending on whether each pixels' information are accessed directly or the frame is processed before the execution of the SLAM algorithm.

2.1.1 Mathematical Formula of SLAM Systems

Given a series of controls and sensor observations over discrete time steps , the SLAM problem [12] is to compute an estimate of the agent's state and a map of the environment . All quantities are usually probabilistic, so the objective is to compute:

$$P(m_{t+1}, x_{t+1} | o_{1:t+1}, u_{1:t})$$

Applying Bayes rule gives a framework for sequentially updating the location posteriors, given a map and a transition function ,

$$P(x_t | o_{1:t}, u_{1:t}, m_t) = \sum_{m_{t-1}} P(o_t | x_t, m_t, u_{1:t}) \sum_{x_{t-1}} P(x_t | x_{t-1}) P(x_{t-1} | m_t, o_{1:t-1}, u_{1:t}) / Z$$

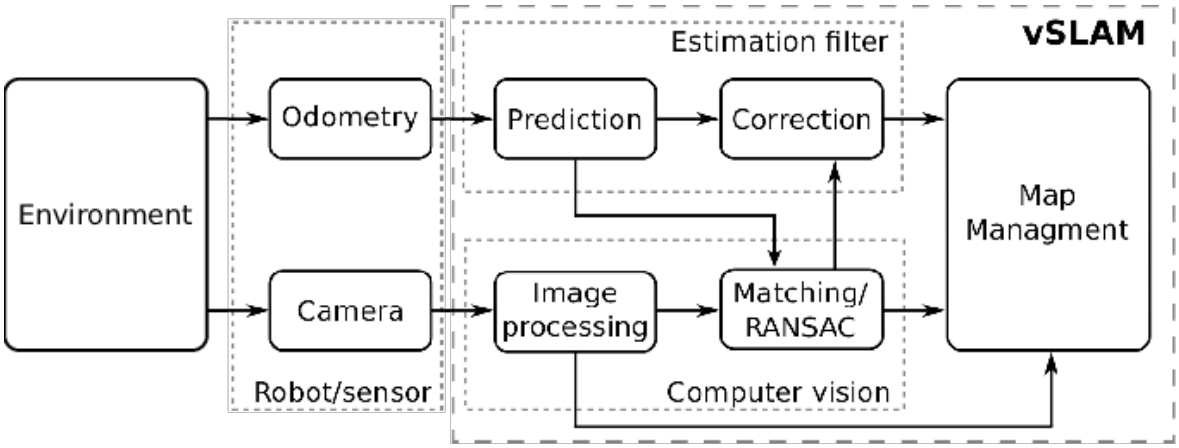
Similarly the map can be updated sequentially by

$$P(m_t|x_t, o_{1:t}, u_{1:t}) = \sum_{x_t} \sum_{m_t} P(m_t|x_t, m_{t-1}, o_t, u_{1:t}) P(m_{t-1}, x_t|o_{1:t-1}, m_{t-1}, u_{1:t})$$

2.1.2 Visual SLAM

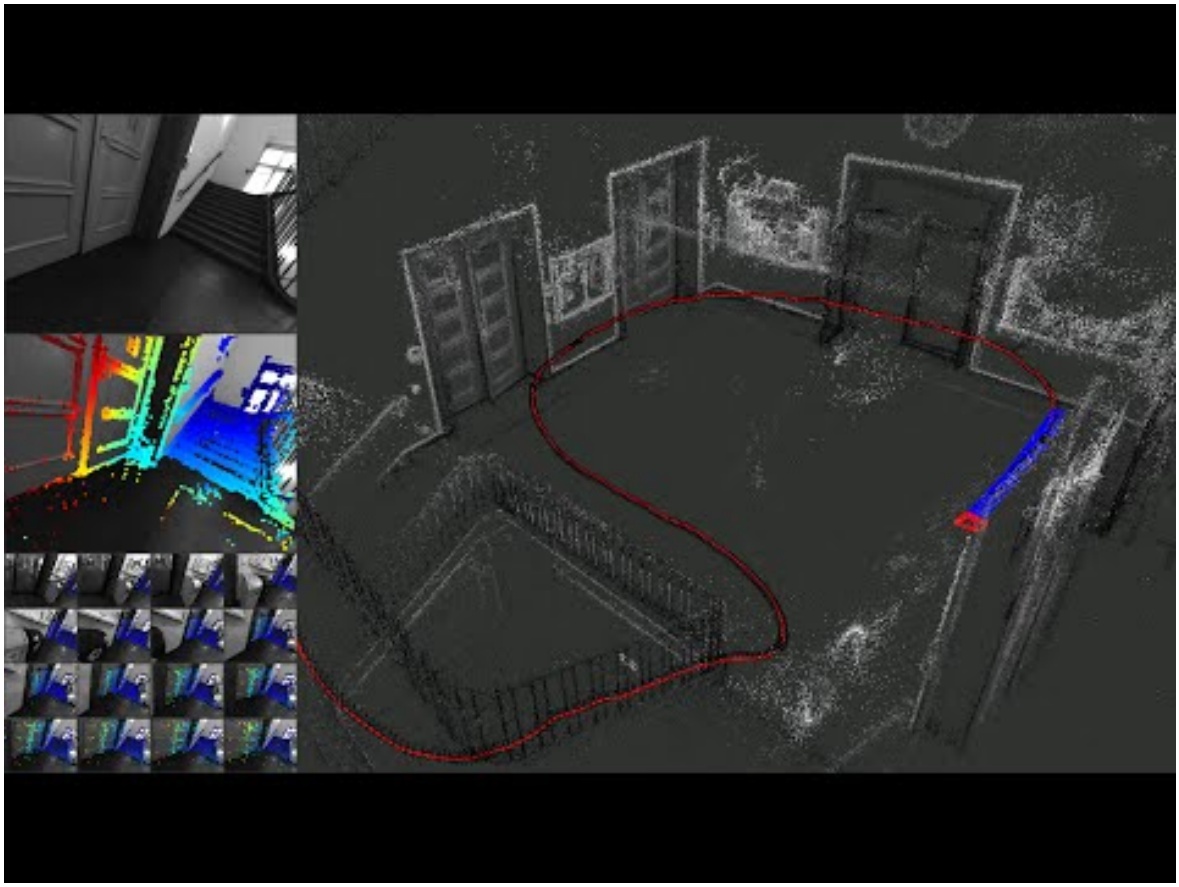
Visual SLAM [10] refers to the process of determining the exact position of a sensor by taking into consideration its surroundings, while mapping the environment around the sensor when neither the environment nor the location of the sensor is known.

The majority of the Visual SLAM Systems [9] function by tracking the 3D position set points through successive camera frames by using the approximate projected position of a camera pose. Unlike to other SLAM Systems, Visual SLAM technology uses single 3D vision camera and often is needed to operate in real-time.



2.1. Monocular visual SLAM

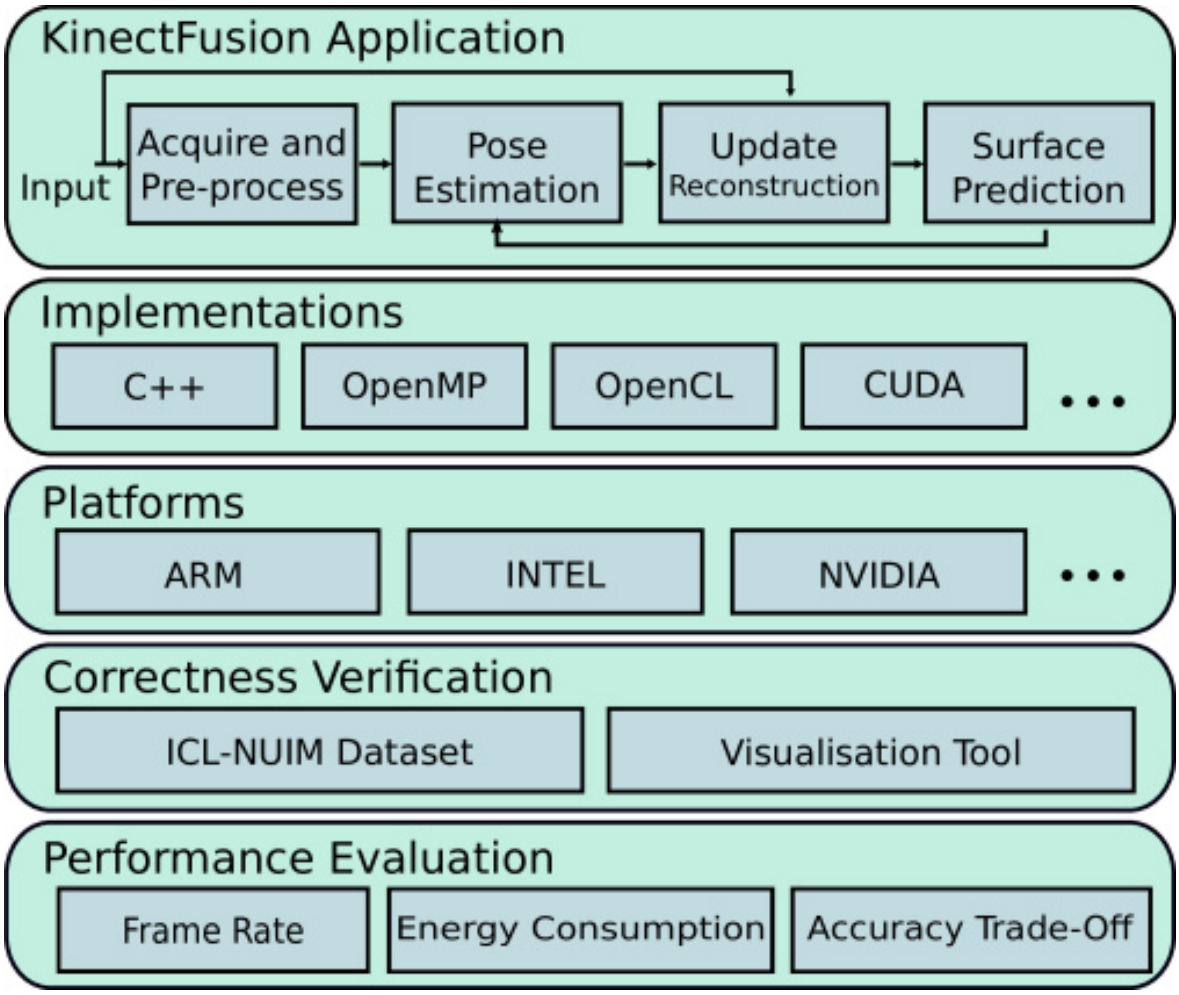
The framework of vSLAM algorithms is composed of five modules: initialization, tracking, mapping, relocalization, and global map optimization. Since each vSLAM algorithm employs different methodologies for each module, features of a vSLAM algorithm highly depend on the methodologies employed. Therefore, it is important to understand each module of a vSLAM algorithm to know its performance, advantages, and limitations.



2.2. DSO: Direct Sparse Odometry

2.2 SLAMbench & KinectFusion

SLAMbench is an OpenSource edition of the Dense SLAM algorithm KinectFusion [13], which is implemented on CUDA, C/C++, OpenMP, and OpenCL and the purpose of its design is the profiling of its performance, the control of its execution and its optimization. SLAMbench [14] operates in the exact same way as the KinectFusion Algorithm, which was designed by Microsoft and which obeys to the rules of a specific process. Although the KinectFusion Algorithm uses as input frames the depth frames extracted from Kinect Motion Controller's camera, the SLAMbench implementation uses as input the ICL-NUIM dataset.



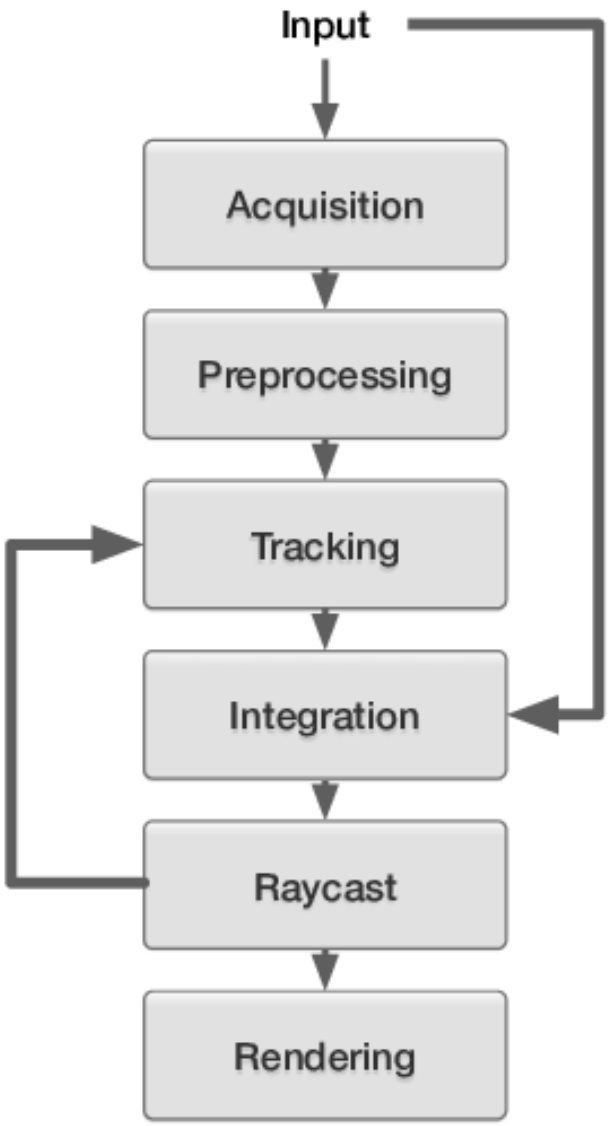
2.3. KinectFusion process

2.2.1 KinectFusion stages

KinectFusion Algorithm's process follows the following stages:

1. **Acquisition** : Input of the depth frame in order to be processed.
2. **Preprocessing** : 2D depth image adjustment and application of a bilateral filter for the elimination of the existing noise. Also during this stage the image is divided in a 3-level image pyramid. Finally, a point cloud is generated and the vertexes are transformed into normals that are used for the position projection using the ICP algorithm.
3. **Tracking** : In this stage a new 3D pose is constructed, by comparing the pose produced in the previous iteration, and the projected pose.

4. **Integration** : Integrates the constructed pose in the 3D map, that we have previously constructed.
5. **Raycast** : A depth frame from the new pose of the camera is produced.
6. **Rendering** :Visualises the 3D reconstruction of the indoor scene that occurred from the execution of the whole algorithm.



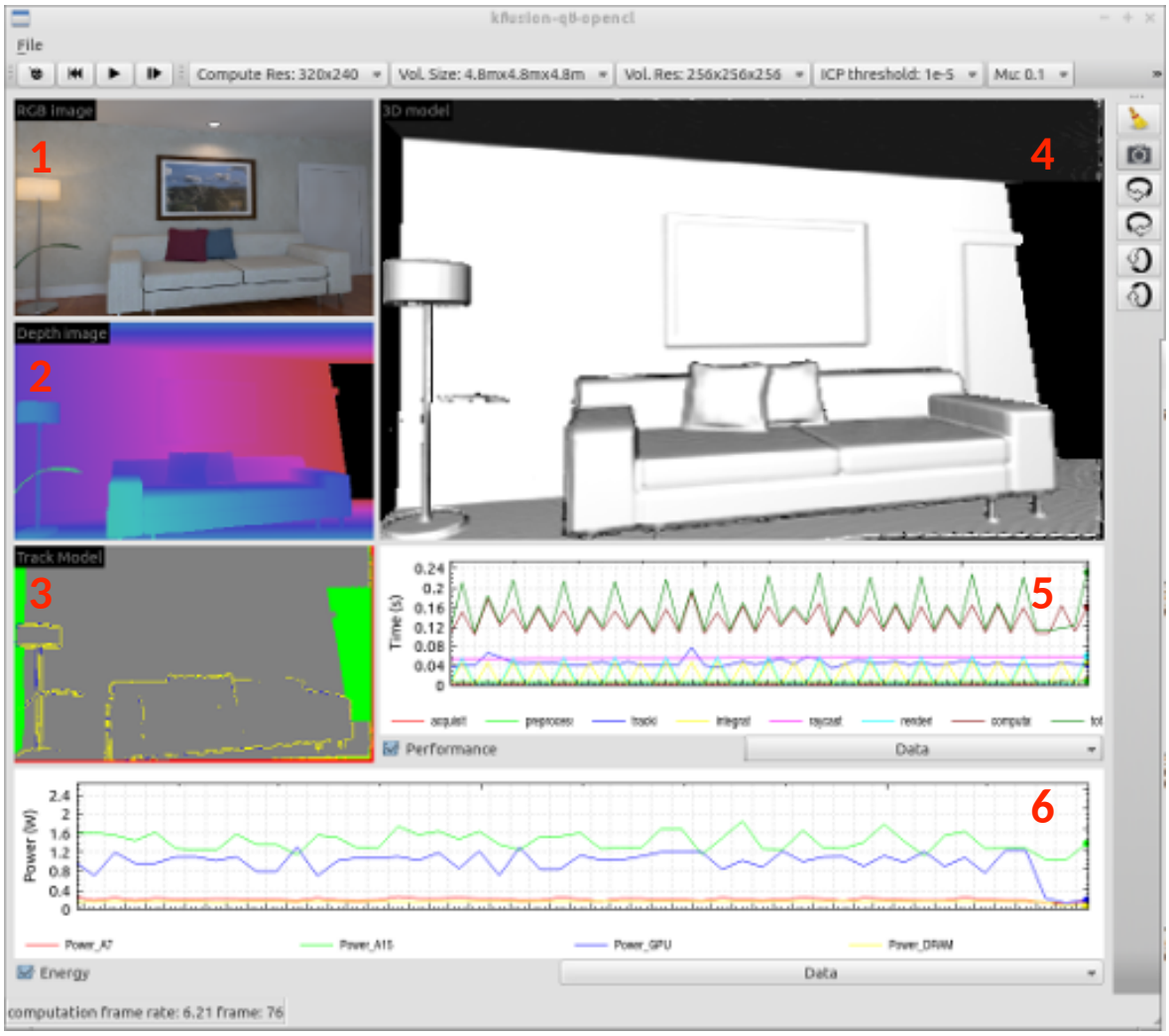
2.4. KinectFusion Stages

2.2.2 SLAMbench kernels

The SLAMbench algorithm's process is completed by the execution of a total of 14 kernels.

1. **acquire**: acquires the new RGB Depth frame and calculates the I/0 costs.
2. **mm2meters**: converts the 2D depth image from millimetres to meters in order to be used by the following kernels.
3. **bilateralfilter**: blurs the depth image so as to reduce the noise and errors.
4. **halfSample**: creates a three-level pyramid. The pyramid levels are used to project the tracking solution of low resolution images, to high resolution ones.
5. **depth2vertex**: transforms each pixel of the depth image into a vertex. A point cloud is generated.
6. **vertex2normal**: computes normal vectors of each vertex of a point cloud, so as to be used by the track kernel.
7. **track**: establishes correspondence between vertices in the synthetic and new point cloud.
8. **reduce**: sums up the distances of corresponding vertices of 2 point clouds for the minimisation process.
9. **solve**: solves a 6x6 linear system, to produce a 6-dimensional vector to estimate the new camera pose.
10. **integrate**: integrates the new produced point cloud into the 3D volume.
11. **raycast**: computes the new point cloud and normals corresponding to the current estimate of the camera position.
12. **renderDepth**: produce a visual construction of the depth map of the depth image captured from the sensor using a colour coding.
13. **renderTrack**: produce a visual contraction of the tracking process' results by colouring each pixel with different colour regarding its accuracy.

14. **renderVolume**: produce a visual 3D reconstruction of the indoor scene, from a specific viewing position.



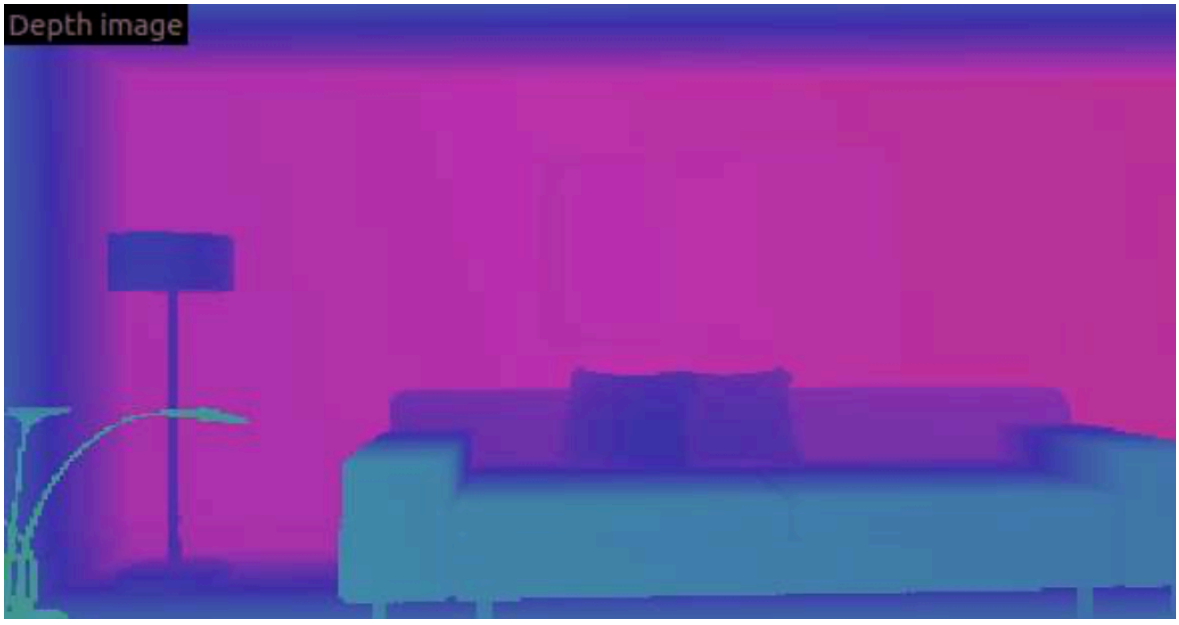
2.5. SLAMbench Execution Results
1) ICL-NUIM RGB-D Image, 2) ICL-NUIM Depth Frame, 3) Tracking Result, 4) 3D Reconstruction after the KinectFusion algorithm's execution 5) Execution Time for KinectFusion Stages, 6) Power consumption on each device.

2.2.3 ICL-NUIM dataset

The SLAMbench algorithms input, is ICL-NUIM dataset [15] which provides the customer with RGB-D depth frames of 2 different indoor scenes, an office and a living room, for the verification of the accuracy of Visual Odometry and SLAM algorithms. The interior of the living room scene contains several common objects like sofa, vase, lamp, art pieces etc. In this diploma thesis, we have used the trajectory lr_kt2 of the living room dataset, which

composes of 882 RGB-D frames. Living room has a 3D surface ground truth together with depth-maps, as well as camera poses and as a result perfectly suits, not just for benchmarking camera trajectory, but also reconstruction.

2.6. ICL-NUIM RGB-D living room



2.7. ICL-NUIM living room depth frame

Chapter 3

3.1 SLAMbench Implementation

3.1.1 Software Implementation

The first step was to execute the KinectFusion Algorithm, on Zynq Ultrascale+ MpSoC ZCU102 ARM processor. In particular the Zynq Ultrascale+ MpSoC contains a quad-core ARM® Cortex-A53 processor, which is a 8-stage pipelined processor with 2-way superscalar, in-order execution pipeline. As mentioned above, the input dataset used for this implementation is trajectory kt2 of the ICL-NUIM living room dataset (lr_kt2) , which contains 882 RGB-D images.

Device	Quad-core ARM Cortex-A53 (included in Zynq Ultrascale+ MpSoC)
Input data	ICL-NUIM living room dataset, trajectory lr_kt2, 882 frames

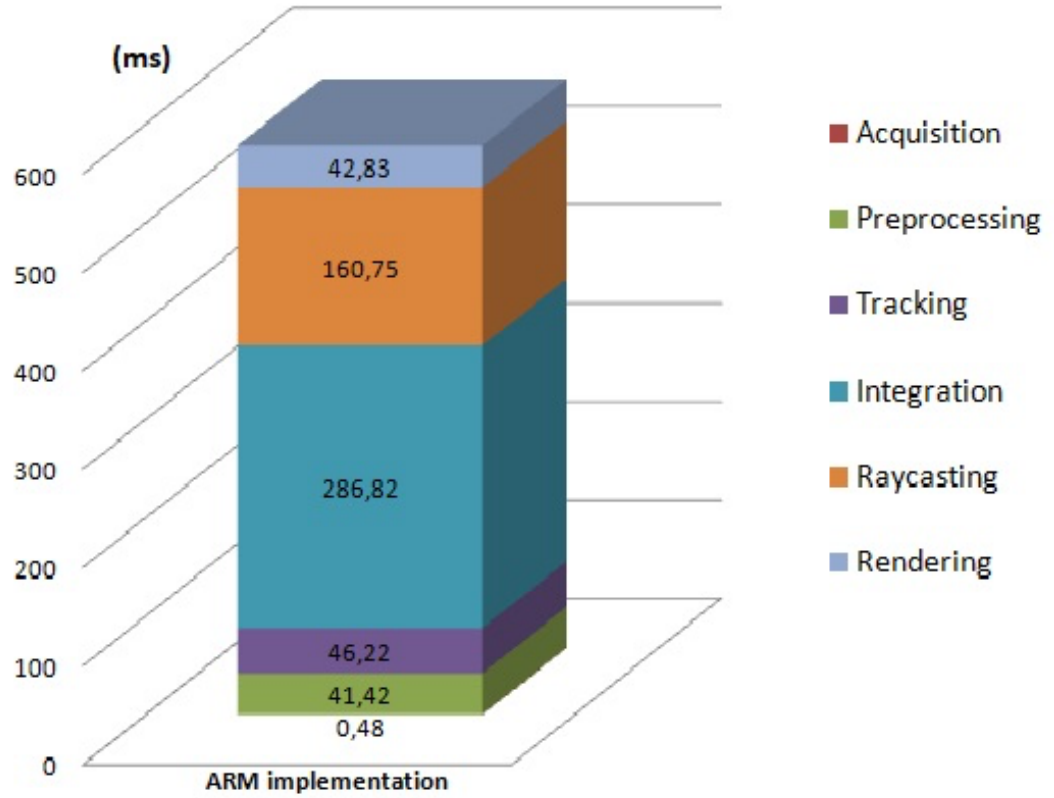
Table 1. General Project Information

The conditions we took under consideration in the implementation of the project, are defined as input parameters to the algorithm . To be more specific:

Parameter	Definition	Value
-p	Initial pose factors	0.34, 0.5, 0.24
-c	Compute Ratio	2
-r	Integration Rate	1
-t	Tracking Rate	1
-z	Rendering Rate	4
-l	ICP-threshold	1E-05
-m	Mu	0.1
-s	Volume Size	4.8
-d	Volume Direction	4
-v	Volume Resolution	256
-y	Pyramid Levels and Iterations of each Level	10, 5, 4

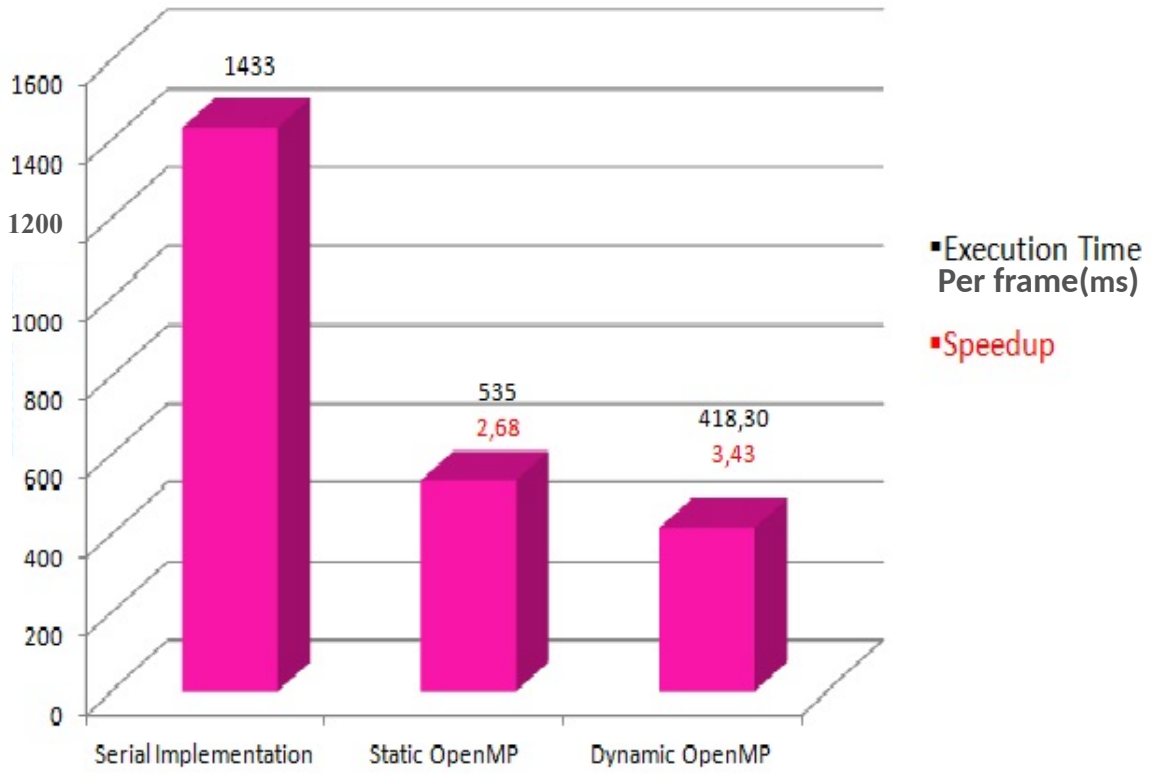
Table 2. Input Execution Parameters

By the single threaded SW execution of the kernels on ARM Cortex-A53, the algorithm runs at 535.7 ms/frame. The following diagram shows the latency of the execution of each stage of the KinectFusion Algorithm, when executed using the aforementioned ARM processor.



3.1. Kernel Time Distribution in KinectFusion per frame

For the parallel execution of the SW algorithm, we used the OpenMP library, so as to distribute the code execution to the processor threads and efficiently utilize the capabilities of the device. By altering the OpenMP functions to static or dynamic, we can see that the dynamic implementation offers a greater optimization. However, in order to carefully observe the contribution of the HW optimizations to the KinectFusion algorithm's latency, we chose the static OpenMP, and the dynamic is used later-on . Thus, by executing the code using the static OpenMP and dynamic OpenMP pragmas, we observe the following results, that show the improvement of the latency and the speedup of the algorithm, using the static and dynamic OpenMP, in comparison to the serial implementation:



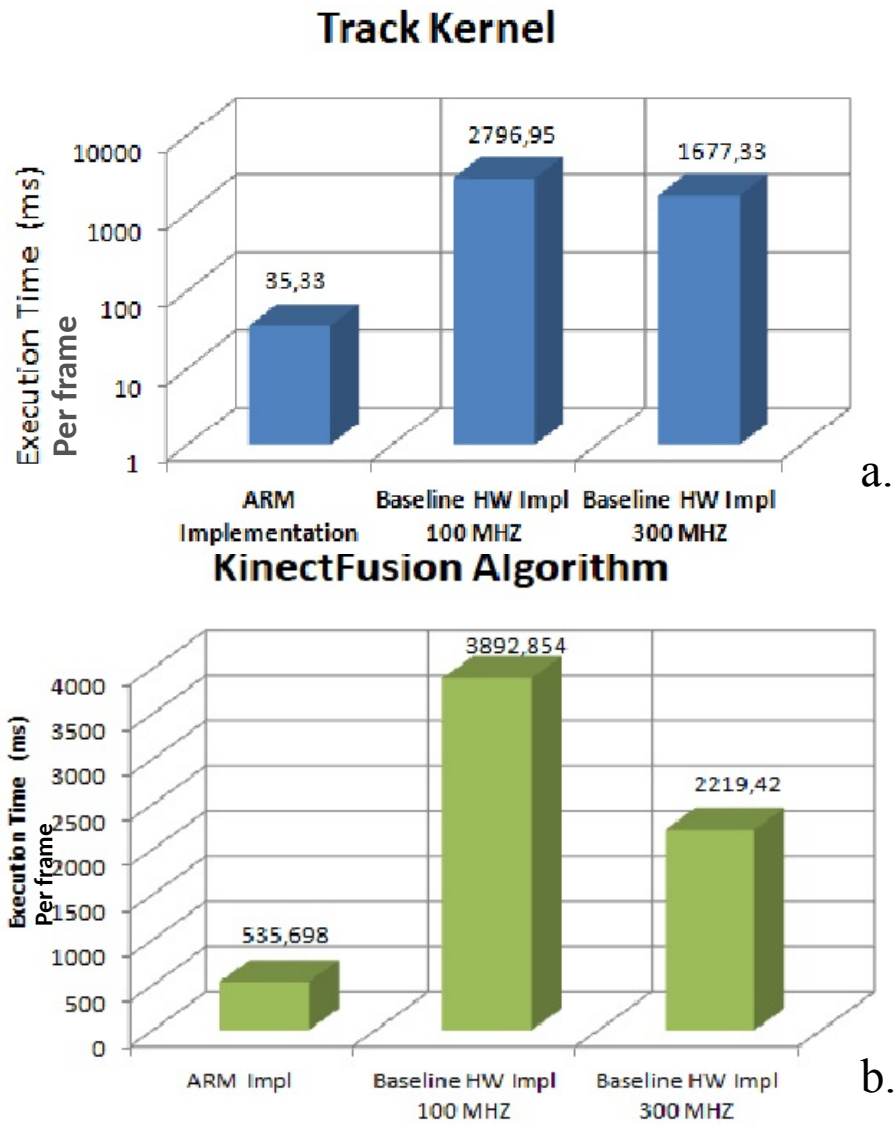
3.2. OpenMP Utilization Time Improvement

3.1.2 Hardware Implementation of the Track kernel

The next step was to implement and accelerate the track kernel in Vivado HLS, in order to profile and optimize its performance. One of the most important stages of KinectFusion algorithm is the tracking stage, which composes of the halfSampleRobustImageKernel, depth2vertexKernel, vertex2normalKernel, trackKernel and the reduceKernel.The track kernel's algorithm begins with the projection of the position of each pixel, depending on the previous position of the same pixel, which was tracked in the previous iteration of the kernel. Afterwards, it calculates the exact position of the pixel to the 3D map, by using the input vertexes and normals, produced in previous kernels, and it stores the values to the output buffers, so as to be integrated in the 3D reconstruction. The output data that the track kernels provides, are an integer value that contains an error for the tracking of each pixel, and 7 output floats that are the elements of the tracked data vector.

The input frames' size is 640x480. However, the resolution of the data inserted to the kernel is reduced to 320x240. Both the input data that we need for the process (normals, vertexes, etc) and the output data, are stored to memory buffers and are accessed in every iteration of the kernel. The execution time of the baseline implementation with 100 MHz clock frequency is 2796.95 ms/frame for the track kernel, 3255.67 ms/frame for the tracking stage and 3892.854 ms/frame for the whole execution of the SLAMbench algorithm.

Nevertheless, the HW execution offers the ability, of using a faster clock, with a higher frequency, thus a smaller clock period. By choosing 300 MHz clock frequency, the baseline implementation of the track kernel runs in 1677.33 ms/frame, the tracking stage runs in 1728.39 ms/frame and the SLAMbench algorithm in 2219.42 ms/frame.



3.3. a) ARM execution and baseline HW implementation of the track kernel and b) the total latencies of the Kinect Fusion algorithm execution

Chapter 4

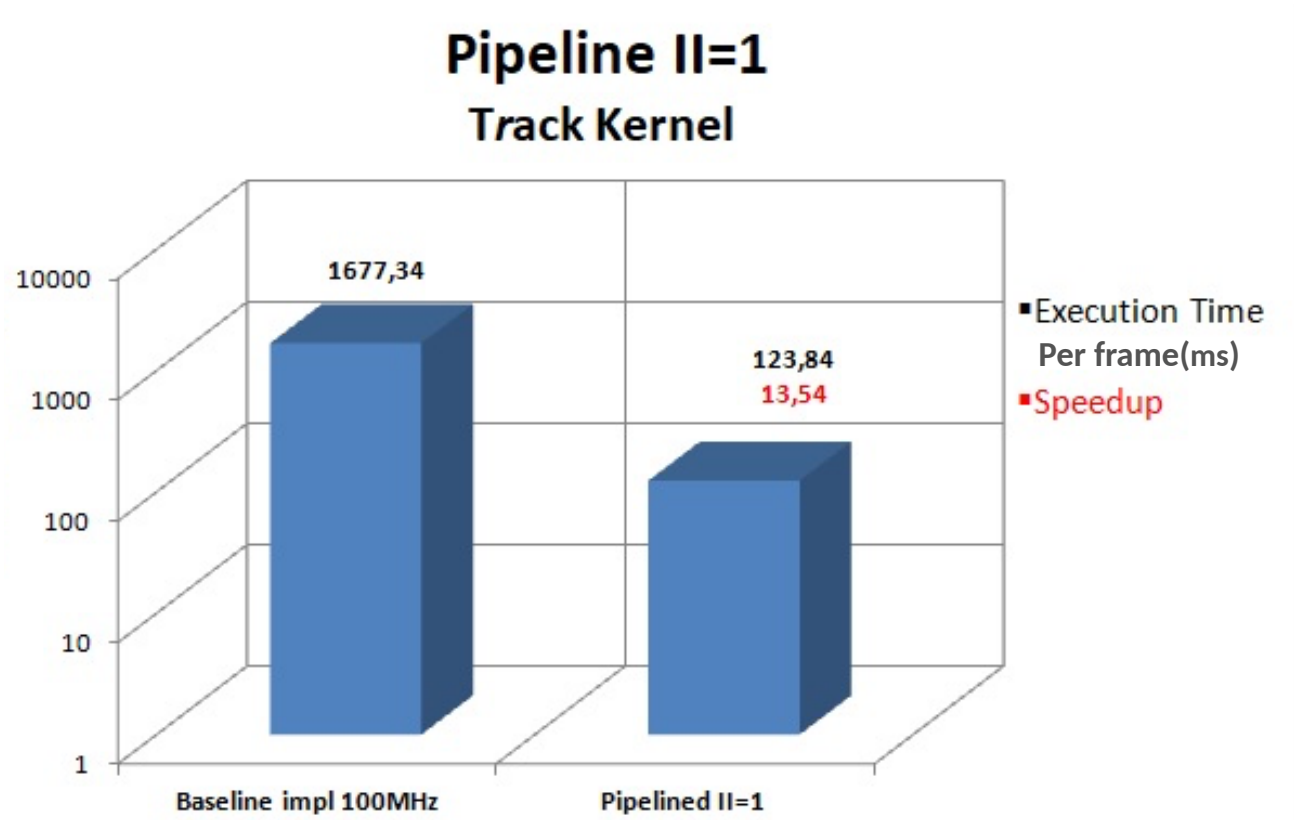
4.1 Optimizing the Track kernel

After the Hardware Implementation of the Track kernel, the execution time, far exceeds the Software Implementation one. So we can notice that this significant difference, renders the choice of HW implementing the kernel, inefficient, and in order to be useful, it should be further optimized.

The first set of optimizations, is precise, so that the accuracy of the algorithm is not affected while applying this kind of optimizations. The Absolute Trajectory Error (ATE) is almost 18 m, and this set of optimizations keep this amount stable. Next we have used some approximate optimizations, so as to further reduce the execution time, though with an of-minor-importance sacrifice of the algorithm's accuracy. Finally, we checked the improvement of the algorithm's speed by adding multiple compute units.

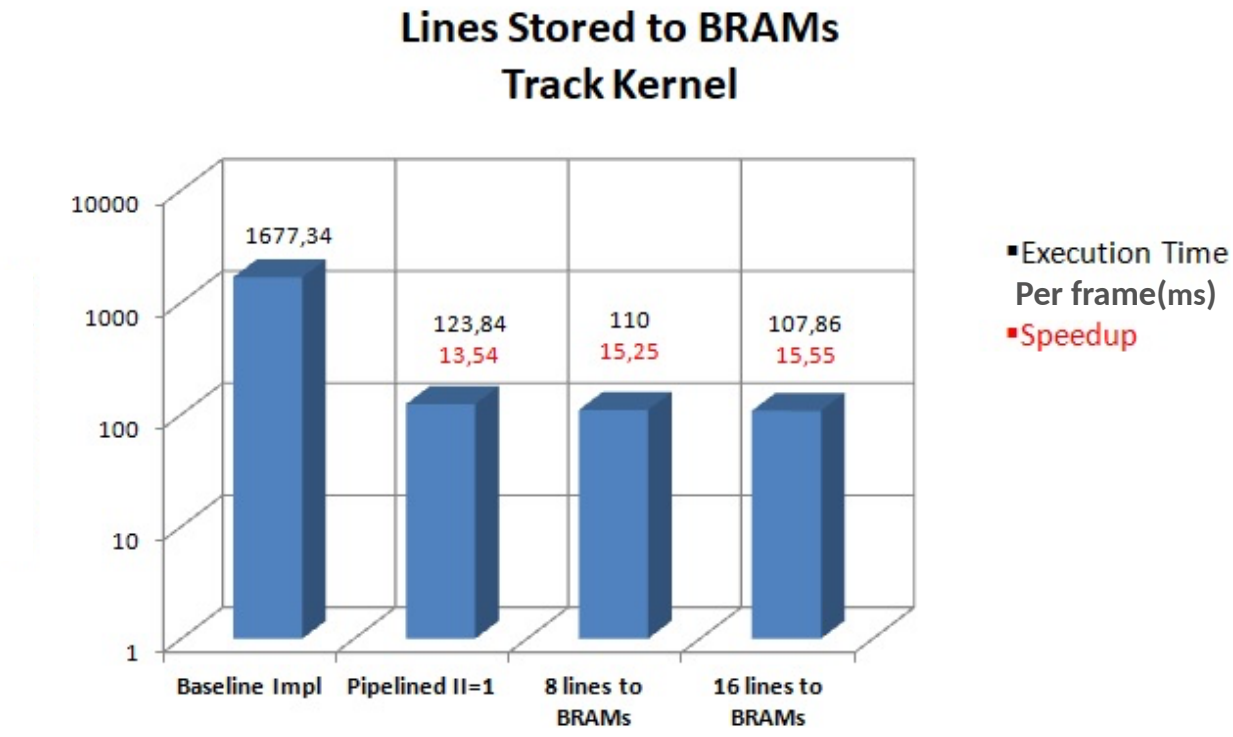
4.1.1 Precise Optimizations

1. **Pipeline:** The first attempt of increasing the performance, is by pipelining the nested for loop, that calculates the exact position of each pixel. Due to the fact that in each iteration of this for loop, more than one accesses in the memory that the output buffer is stored, occur, during the storing process of the output buffer's values, a line of the output buffer is stored in 7 Block RAMs (BRAMs), which are partitioned in a cyclic way, so that each memory position of the output buffer can be accessed in the same clock cycle. With the addition of some inner computational changes inside the for loop, we accomplished to pipeline it with II = 1 (Iteration Interval = 1). This change resulted to a significant decrease of the latency of the track kernel as it reached the 123ms/frame. This latency includes also some other optimizations such as the deletion of the unused registers, and calculations that happen more than once etc. The following diagram shows the track kernel's latency results, and the speedup of the pipelined implementation, compared with the baseline implementation (300 MHz clock frequency).



4.1. Baseline 300MHz and Pipeline Latency
ATE = 18m

2. Multiple Lines Stored to BRAMs: The second precise optimization is transferring and storing more than one lines of the output buffer to Block RAMs so as to reduce the memcpy function revision, that copies the values to the output buffer. I have chosen to use 8 and 16 lines, and the result of the latency of the track kernel is 110ms in 8 lines storage, and 107.86 ms/frame in 16 lines storage. The diagram below demonstrates all the optimizations so far, and the speedup , in comparison to the baseline implementation. The ATE of this optimization remained almost 18m.

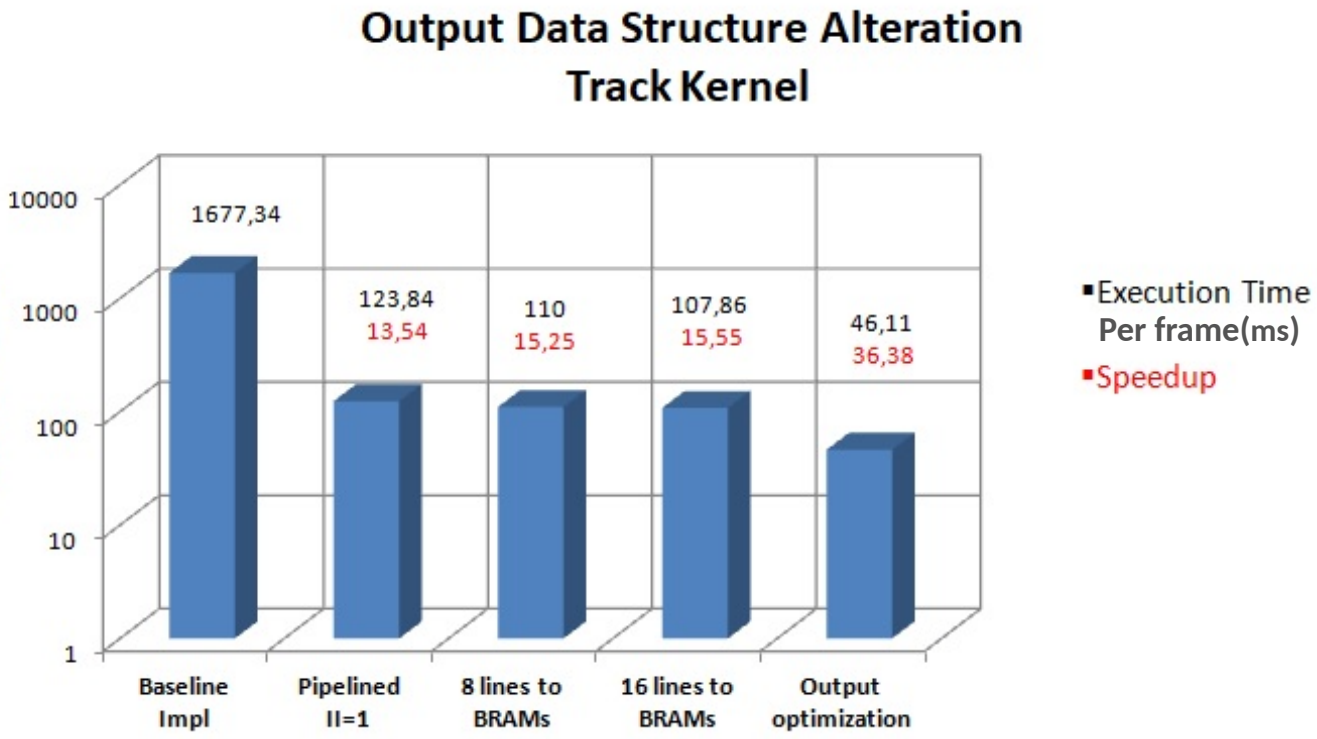


4.2. Multiple Lines Stored to BRAMs Latency and Speedup
ATE = 18m

3. Unrolling: The inner nested for loop was manually unrolled and pipelined in order to achieve more parallel accesses and the simultaneous execution of the tracking process for 2 pixels. The desire to accomplish a manual unrolling was satisfied by importing the normal and vertex buffers twice so as to simultaneously access them and by changing the structure of the temporary output buffer, to a 2D buffer (7x320). Finally the output buffer was stored in 14 BRAMs as each clock cycle we have 14 store operations to the output buffer. However, because of the increased number of memory accesses and memcpy function calls that optimization did not help and the latency significantly increased.

4. Output data structure alteration: The last precise optimization implemented to the code of the previous optimization (16 lines are stored to BRAMs), is the alteration of the output buffer's elements' type. At first, the kernel's output data was an integer and 7 floats for the iteration for each pixel. So these 8 values, were combined in 2 float4 type registers, so that less BRAMs are used and only 2 accesses are now happening in each iteration.

After the execution of the kernel including all the above optimizations, the final execution time reached the 46.11 ms/frame for the track kernel, 63.17 ms/frame for the tracking stage and 545.38 ms/frame for the SLAMbench algorithm.



4.3. Latency and Speedup of the Output Data Structure Alteration compared to the previous optimization results

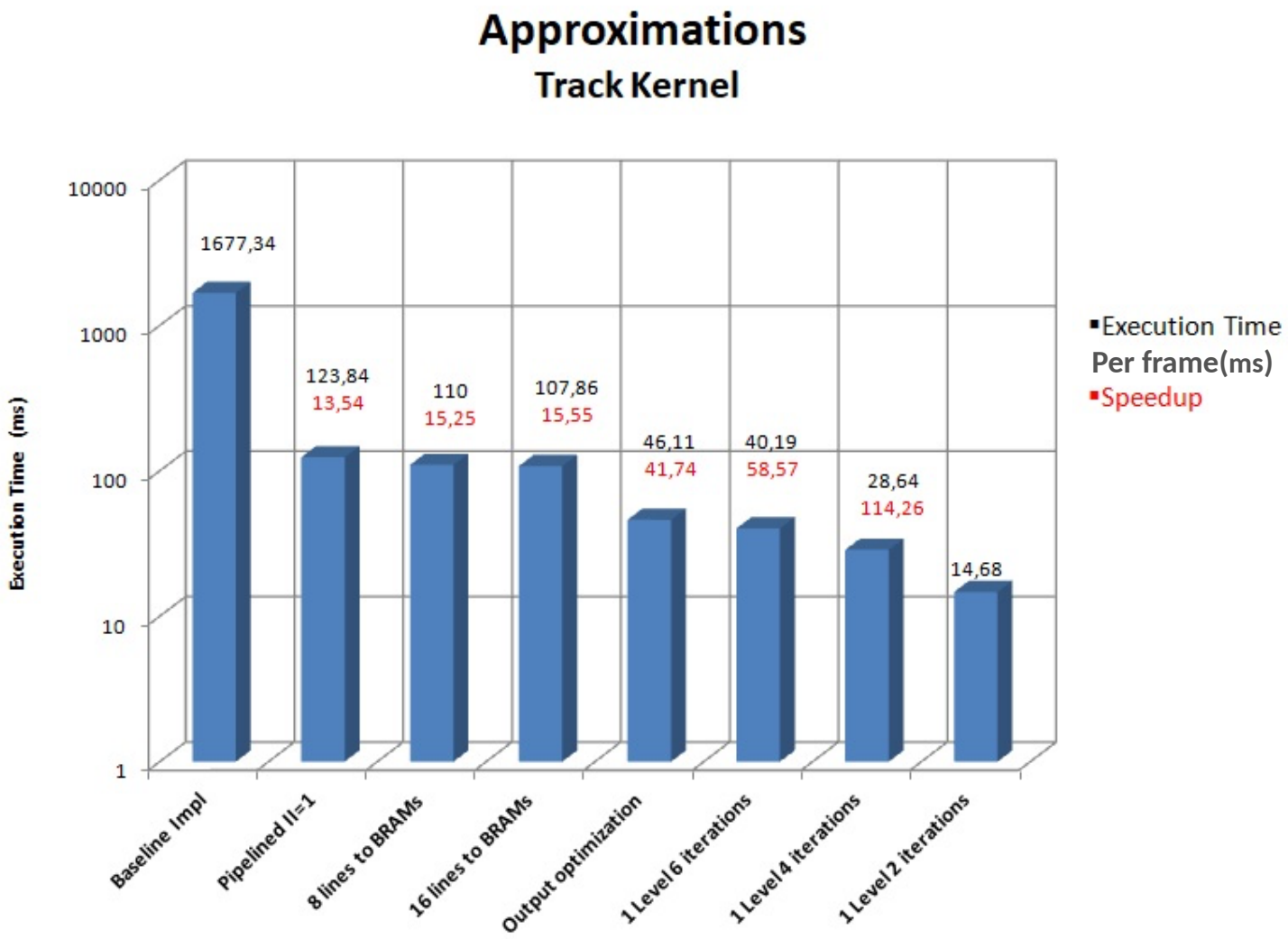
ATE = 18m

4.2 Approximate Optimizations

1. **Pyramid Levels Reduction:** As mentioned before, the input depth frame is divided in 3 pyramid levels so every execution of the track kernel addresses to a different level, and the more the levels, the higher the tracking process' accuracy. So, the reduction the pyramid levels to 1 level, lead to the reduction of the latency but the fact that the ICL-NUIM RGB-D frames do not have significant noise, we observe no increase of the ATE of the algorithm.
2. **Kernel's Iterations Reduction:** Apart from the decrease in the number of levels, we decreased the number of iterations of the execution of the track kernel. In the initial implementation, in order for the noise to be eliminated and for the accuracy of the results to be guaranteed, the kernel is executed more than once for each input depth frame. In the baseline implementation each the first level of the pyramid was executed 10 times, the second level 5 times and the third 4 times. However, as the input depth frames of the trajectory lr_kt2 of the ICL-NUIM dataset , do not have significant noise, we are allowed to iterate the kernel's execution less times, without losing any part of the accuracy of the algorithm. So for checking purposes, we sequentially reduced the number of iterations in 6, 4 and 2. Although the kernel is executed less times for each frame, the ATE remains almost the same with the initial approach (ATE = ~18 - 19 m).

After the approximations the can se a significant decrease of the execution time as for the track kernel is 14.68 ms/frame, for the tracking stage is 23.48 ms/frame and for the SLAMbench algorithm is 505.99 ms/frame.

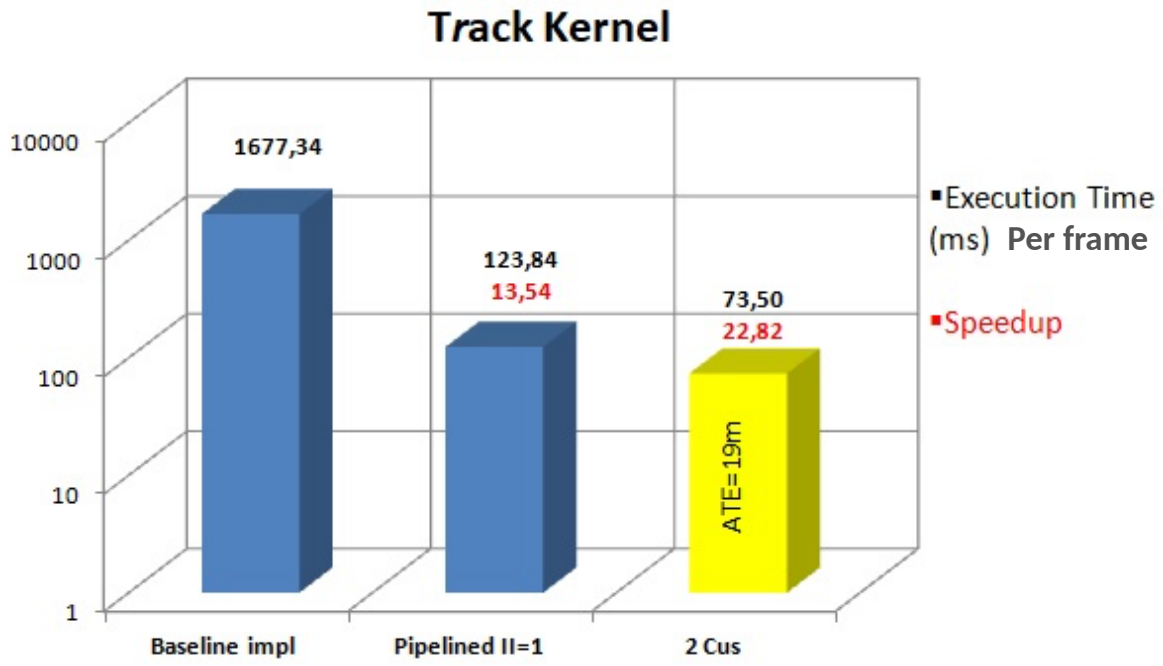
To the following diagram we can notice the results after the implementation of the approximate optimizations. Latency of each optimization and the speedup (in dependance to the baseline implementation) are shown below:



4.4. Approximate Optimizations Latency and Speedup in comparison with the Previous Optimization Results
ATE = 18 - 19m

4.3 Multiple Compute Units

The last optimization that was implemented, is the use of multiple compute units, that are executed simultaneously, and each one processes a part of the depth image. By using 2 parallel compute units (each CU executes the tracking algorithm on the half depth frame), after the first precise optimization (Pipeline II=1), the latency was insignificantly improved (73.63ms for the execution of the track kernel). Nevertheless, by adding 2 compute units in that implementation, due to the process that we have followed, the ATE increased a little bit (ATE = 19m), which is not a significant change.

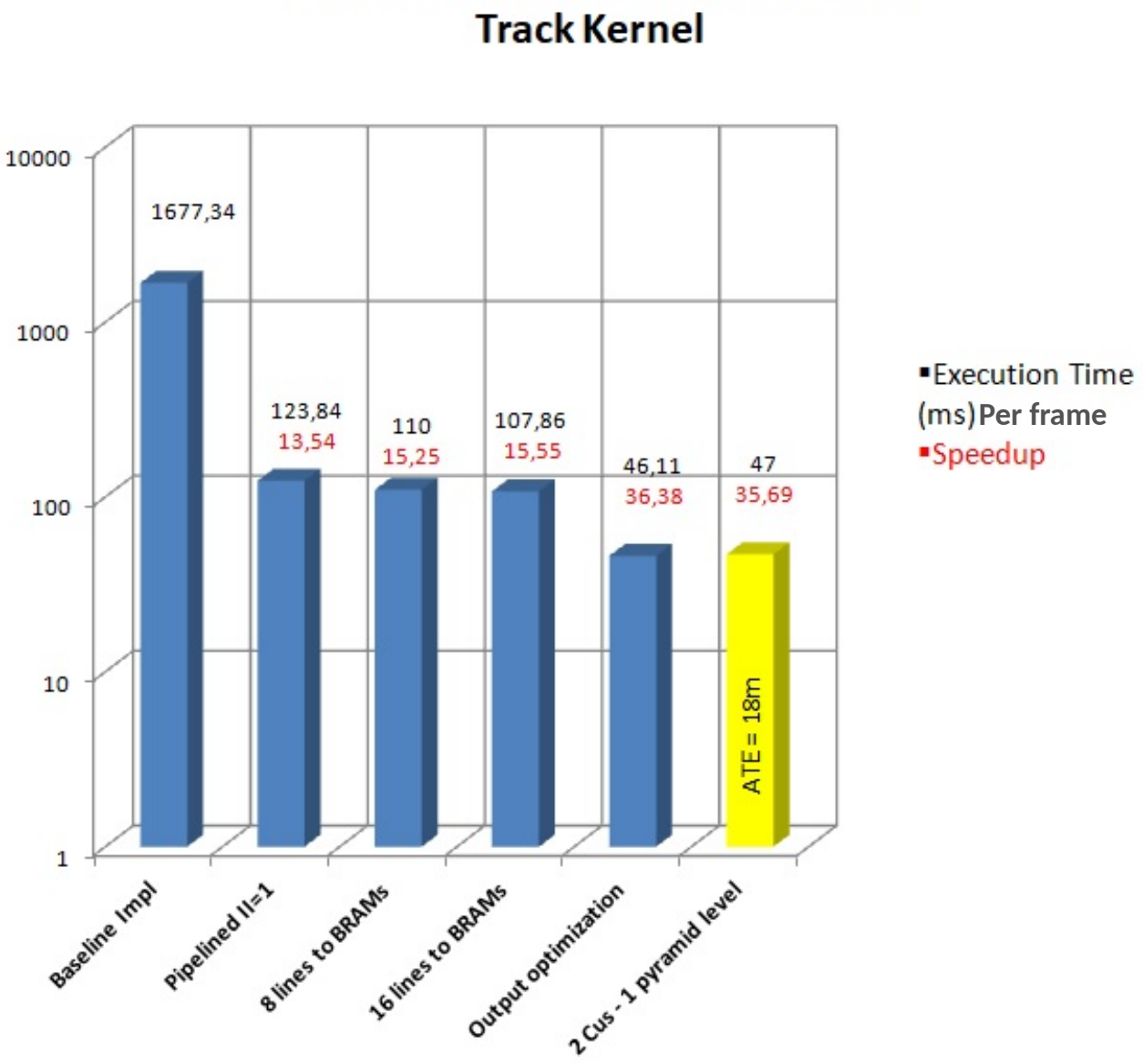


4.5. Multiple Pipelined Compute Units Latency compared to the Pipelined and Baseline Implementation

The second application of the 2 CUs that are executed in parallel, was implemented in the first approximate optimization, which was the reduction of the pyramid levels to 1 level. Although ATE remained stable (ATE = 18m), the latency of the execution of the 2 CU reached the 47 ms/frame, which is higher than the simple CU implementation.

Next, we tried to use sub buffers for the output data, instead of dividing the output buffer to 2 half buffers, but the execution timing result was insignificantly worse (54 ms/frame for the execution of the 2 CUs).

In other words, either the compiler, in order to maintain the accuracy of the algorithm, take more time to execute it, or when it tries to execute it faster, the accuracy is sacrificed. That happens because when each CU is executed, a minor mistake occurs. Thus, when added 2 more CUs (4 CUs in total), or when we tried to apply 2 CUs after the end of the precise and approximate optimizations, the accuracy was completely lost, leading to the existence of numerous untracked frames, and a huge ATE number. Consequently, this implementation cannot be used, but also it cannot provide us with a more efficient algorithm, as the existence of more than 1 compute units did not benefited the whole process. As we can notice from the 2 graphs, the use of multiple compute units is not the optimal solution.



4.6 Multiple Optimized Compute Units Latency compared to the Previous Implementations

4.4 Area utilization

The last thing that we should mention is the area occupation of each implementation. As this Thesis is concentrated in the latency reduction, the area utilization is not optimized, but it would be necessary to observe the transformation of the available space of the Zynq Ultrascale+ MpSoC. Thus, we collected the information about the BRAM, DSP, FF and

LUT utilization, during the best precise optimization, the final approximate optimization and the optimal multiple CU implementation, and presented them in the following table:

	BRAM(%)	DSP(%)	LUT(%)	FF(%)
Baseline Implementation 300 MHz	1.14	3.02	6.81	5.03
Optimal Precise Implementation	6.04	8.25	11.94	8.44
Optimal Solution	6.04	8.25	11.94	8.44
2 CUs Implementation	8.74	16.75	24.38	17.3

Table 3. Multiple Optimized Compute Units Latency compared to the Previous Implementations

So we can notice that, as usual, as the time decreases the area is more and more utilized.

Chapter 5

5.1 Conclusion

In the final step of the process of this Thesis, we should examine if our implementation offers an optimized solution for the execution of the KinectFusion algorithm. In fact, the HW execution of the track kernel, provides a higher speed than the ARM implementation as shown in the following figure. For this reason, we have accomplished a faster kernel, that helps to the reduction of the complete execution time per frame of the KinectFusion algorithm, while not affecting the algorithm's error.

5.1. ARM Kernel Implementation Latency per frame and Optimized FPGA Execution Latency and Speedup

However, we can notice that the track kernel is not the ideal algorithm to be optimized, as the HW implemented kernel's latency is not dramatically lower than the latency of the kernel execution on ARM. Therefore, any latency improvement is helpful, when trying to optimize the execution time of an algorithm.

5.2 Future work

The last question that we should pose, in order to accomplish a fully optimized SLAMbench process is whether we could accelerate all the other kernels of the SLAMbench in Vivado HLS, using the ZCU102 FPGA, optimize their operation, and finally incorporate all the hardware executed kernels, so that we can refer to a fully optimized KinectFusion algorithm.

Bibliography

1. Wikipedia, “Field-programmable gate array” URL: *https://en.wikipedia.org/wiki/Field-programmable_gate_array*

2. Xilinx Team, “ZCU102 Evaluation Board”, *UG1182 (v1.6) June 12, 2019*. URL: *https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf*

3. Xilinx Team, "Vitis Unified Software Platform Documentation, Embedded Software Development”, *UG1400 (v2019.2) March 18, 2020*. URL: *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1400-vitis-embedded.pdf*

4. Xilinx Team, “Vitis Unified Software Platform Documentation, Application Acceleration Development”, *UG1393 (v2019.2) February 28, 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1393-vitis-application-acceleration.pdf*

5. Xilinx Team, "Vivado High-Level Synthesis,Accelerates IP Creation by Enabling C, C++ and System C Specifications”. URL: *https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html*

6. Wikipedia, “OpenCL” URL: *https://en.wikipedia.org/wiki/OpenCL*

7. Munshi, Aaftab. "The opencl specification." *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 2009.

8. Stone, John E., David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." *Computing in science & engineering* 12.3 (2010): 66-73.

9. Vision Online Marketing Team, “What is Visual SLAM Technology and what is it Used For?”, *05/15/2018. URL: https://www.visiononline.org/blog-article.cfm/What-is-Visual-SLAM-Technology-and-What-is-it-Used-For/99*

10. Taketomi, Takafumi, Hideaki Uchiyama, and Sei Ikeda. "Visual SLAM algorithms: a survey from 2010 to 2016." *IPSJ Transactions on Computer Vision and Applications* 9.1 (2017): 16.

11. Dong-Won Shin, “Introductory Level SLAM Seminar”, *Jan. 18, 2018*. URL: *https://www.slideshare.net/DongWonShin4/introductory-level-of-slam-seminar?next_slideshow=1*

12. Wikipedia, “Simultaneous Localization and Mapping”. URL: *https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping*

13. Newcombe, Richard A., et al. "KinectFusion: Real-time dense surface mapping and tracking." *2011 10th IEEE International Symposium on Mixed and Augmented Reality*. IEEE, 2011.

14. Nardi, Luigi, et al. "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM." *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015.

15. Handa, Ankur, et al. "A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM." *2014 IEEE international conference on Robotics and automation (ICRA)*. IEEE, 2014.