UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Precise and Approximate Optimizations for Visual SLAM on GPUs

Diploma Thesis

Pavlos Aimoniotis

Supervisor: Nikolaos Bellas

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Precise and Approximate Optimizations for Visual SLAM on GPUs

Diploma Thesis

Pavlos Aimoniotis

Supervisor: Nikolaos Bellas

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ακριβείς και Προσεγγιστικές Βελτιστοποιήσεις του Visual SLAM σε GPUs

Διπλωματική Εργασία

Παύλος Αιμονιώτης

Επιβλέπων: Νικόλαος Μπέλλας

Βόλος 2021

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos D. Antonopoulos**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 16-6-2021

Acknowledgements

I would like to thank my thesis supervisor Prof. Nikolaos Bellas, not only for his great guidance during my research and writing of this paper, but also for our collaboration all these years. Prof. Bellas was always available for me and always steering me to the right direction. For these, I would like to express him my deepest appreciation and huge thanks.

To PhD Candidate of CSL Laboratory, Maria Rafaela Gkeka, thank you for always being available for me, and for being extremely patient with my impatience!

To my family and friends, thank you for the support and for encouraging me throughout my five years of study at the University of Thessaly. Thank you for understanding me and for always being there for me! This accomplishment would not have been possible without you.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Pavlos Aimoniotis
16-6-2021

Abstract

Simultaneous Localization and Mapping (SLAM) is the problem of creating or updating a map of an unknown environment when monitoring an agent's position within it. SLAM algorithms are used in navigation, robotic mapping, and odometry for virtual reality and augmented reality, and they are still a heavily researched subject. Where sensor input is inadequate to evaluate the environment and position, visual SLAM is a type of SLAM system that uses 3D vision to perform location and mapping. The basic aim is to map their surroundings in relation to their current location.

SLAM algorithms have high computational and energy requirements which make implementations very challenging. In this thesis, we propose a high-performance SLAM implementation for GPU devices using OpenCL framework. Our work introduces different optimization techniques, for both precise and approximate optimization of KinectFusion, a well-known SLAM system. We show that proper approximations can enable high performance at almost 634 frames per second, using NVidia GeForce GTX 770 Graphics Processing Unit (GPU), with high energy efficiency and without compromising agent tracking and map construction.

Περίληψη

Το Simultaneous Localization and Mapping (SLAM) αναφέρεται στο πρόβλημα κατασκευής ή ενημέρωσης ενός χάρτη ενός άγνωστου περιβάλλοντος, παρακολουθώντας παράλληλα την τοποθεσία ενός πράκτορα μέσα σε αυτό. Το SLAM παραμένει ένα ενεργό ερευνητικό θέμα και οι αλγόριθμοι χρησιμοποιούνται στην πλοήγηση, τη ρομποτική χαρτογράφηση και την οδομετρία για εικονική πραγματικότητα (virtual reality) ή επαυξημένη πραγματικότητα (agumented reality). Το Visual SLAM είναι ένας τύπος συστήματος SLAM που αξιοποιεί την τρισδιάστατη όραση για τη θέση και τη χαρτογράφηση όταν οι πληροφορίες του αισθητήρα δεν επαρκούν για τον προσδιορισμό του περιβάλλοντος και της θέσης. Βασικά, ο στόχος είναι να χαρτογραφήσουν το περιβάλλον τους σε σχέση με τη δική τους τοποθεσία.

Οι αλγόριθμοι SLAM έχουν υψηλές υπολογιστικές και ενεργειακές απαιτήσεις που καθιστούν τις υλοποιήσεις πολύ δύσκολες. Σε αυτή τη διατριβή, προτείνουμε μια εφαρμογή SLAM υψηλής απόδοσης για συσκευές GPU που χρησιμοποιούν το πλαίσιο προγραμματισμού OpenCL. Η δουλειά μας εισάγει διαφορετικές τεχνικές βελτιστοποίησης, τόσο για την ακριβή όσο και κατά προσέγγιση βελτιστοποίηση του KinectFusion, ενός γνωστού συστήματος SLAM. Δείχνουμε ότι οι σωστές προσεγγίσεις μπορούν να επιτρέψουν υψηλή απόδοση σε σχεδόν 634 καρέ ανά δευτερόλεπτο, σε επεξεργαστή γραφικών NVidia GTX 770, με υψηλή ενεργειακή απόδοση και χωρίς συμβιβασμούς στην παρακολούθηση των πρακτόρων και την κατασκευή χαρτών.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Visual SLAM is becoming increasingly relevant in recent technology. Industries like robotics and navigation focus deeply on it. Visual SLAM is not concerned about any specific algorithm or piece of software. It's a technique for assessing a sensor's location and orientation in relation to its surroundings while simultaneously imaging the world around it.

Most systems map fixed points through successive camera frames to triangulate their 3D position, while simultaneously using this information to approximate camera pose. Basically, the goal of these systems is to monitor their surroundings in comparison to their own position for navigation purposes. To solve this problem, most systems use *Sparse* SLAM algorithms, which preserve key points while reducing computational requirements, and are typically limited to localization. In the other hand, the *Dense* SLAM algorithms use all pixels in an input frame and have the potential to model the 3D scene more thoroughly. Nevertheless, the high requirements in computation and resources make implementation very difficult.

We propose a high-performance KinectFusion implementation on OpenCL framework. We combine precise computing techniques and approximate computing techniques, to achieve excessive performance on an NVidia GeForce GTX 770 Graphics Processing Unit.

1.1 Content Overview

Chapter 2 describes the fundamentals. Provides information crucial to continue reading this thesis.

Chapter 3 presents all the optimizations we applied into KinectFusion. There are two different techniques, precise optimizations and approximate optimizations.

Chapter 4 illustrates the experimental results.

Chapter 5 concludes the thesis.

Chapter 2

Background

In this chapter we will go through the fundamentals. We will see how SLAM algorithms work, details of KinectFusion, what is OpenCL framework and we will describe the NVidia GeForce GPU.

2.1 Simultaneously Localization and Mapping

2.1.1 Mathematical description

Given a series of controls u_t and sensor observations o_t over discrete time steps t, the SLAM problem is to compute an estimate of the agent's state x_t and a map of the environment m_t [8]. All quantities are usually probabilistic, so the objective is to compute:

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

Provided a map and a transformation function, Bayes' rule provides a mechanism for updating the position posteriors sequentially $P(x_t|x_{t-1})$,

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

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 Systems

Slam systems [9, 2, 10, 11] consist of two parts:

1. *Mapping:* create a map of the environment
2. *Localization:* Identifying the agent's relative location and orientation in the world

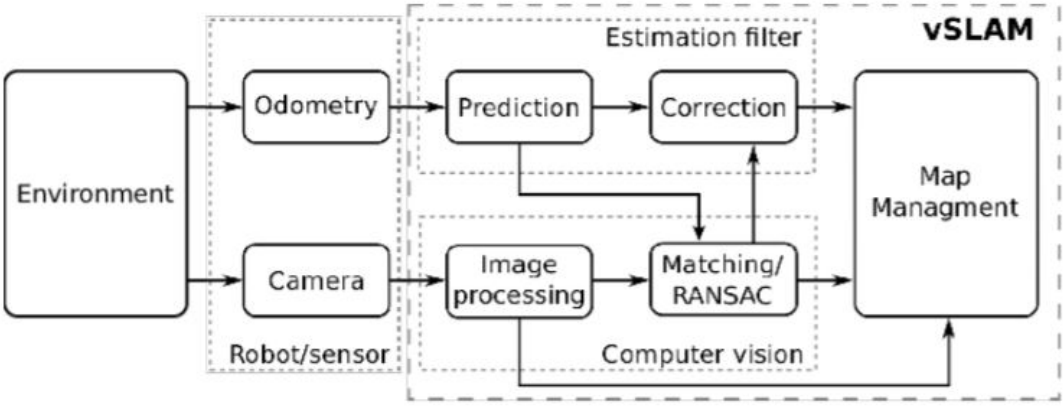

Figure 2.1: Monocular Visual SLAM [1]

Many different sensor types are used by SLAM systems, and therefore depending on the sensor, algorithms that are applied are also different. Exteroceptive sensors offer definitions of several points in an area, removing the need for SLAM inference because shapes can be conveniently and unambiguously aligned by image registration at each level in these point clouds. Tactile sensors are extremely sparse since they only have knowledge on points very close to the agent, so they require strong previous models to compensate for only tactile SLAM. Many practical SLAM events fail somewhere between these graphic and tactile peaks.

The following sensors are used in modern technology:

- Acoustic Sensors
- Visual Sensors
- Laser Range Finders

Visual SLAMs [12] use visual sensors, including RGB-D cameras and monocular cameras. Unlike other SLAM implementations, Visual SLAM depends on a single 3D vision camera, which is supposed to function in real time. In robotics and computer vision, visual odometry is a way of determining a robot's position and path by analyzing the accompanying camera images.

Finally, structures are categorized as sparse or dense. SLAM systems that are considered *Sparse* only use a small subset of the pixels in an input frame. On the other hand, when all pixels of a frame are taken in consideration, we consider those systems as *dense*.

Figure 2.2: Sparse System (MonoSLAM) [2]

Figure 2.3: Dense System (KinectFusion)[3]

2.1.3 KinectFusion

KinectFusion [3, 13] is a variable lighting system for the precise real-time mapping of complex and arbitrary indoor scenes, and is used for dense surface mapping and localization. It was introduced by Microsoft in 2011, and since then it is considered one of the most used algorithms for SLAM systems.

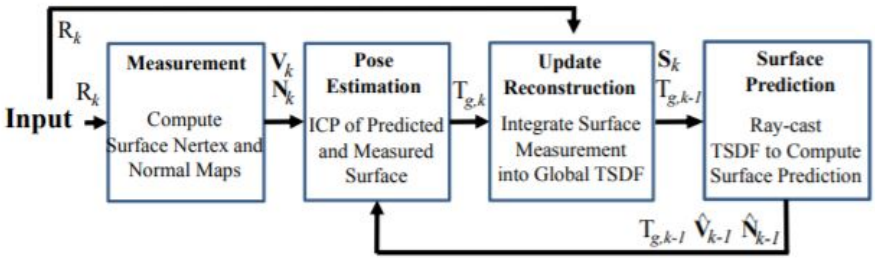

Figure 2.4: KinectFusion System Workflow [4]

In this research, we use SLAMBench [5], a freely accessible software tool that targets benchmarking of SLAM systems. It is used for quantitative, comparable and validated experimental analysis to study trade-offs in the performance, accuracy and energy consumption of a dense RGB-D SLAM device. SLAMBench includes KinectFusion's C++, OpenMP[14], OpenCL[15] and CUDA implementation, reffered as KFusion, and uses the ICL-NUIM dataset of simulated trajectory and scenario ground truth RGB-D sequences for detailed precision comparison of different algorithms and implementations.

The KinectFusion[3] method has six stages, but some of them are broken into multiple kernels in SLAMBench's KFusion. KinectFusions pipeline, as well as KFusions Kernels are shown below.

1. **Acquire** A new RGB-D frame is obtained. This phase is specifically included in order to account for I/O costs during benchmarking and in real-world implementations.

2. **Reprocess**

    - Kernel 1: **mm2meters** from millimeters to meters, a 2D depth image is converted. Just a portion of the image is transformed and mapped into the output if the input image size is not the usual 640x480.
    - Kernel 2: **bilateralFilter** is a depth picture blurring filter that protects the corners. It mitigates the consequences of noise and inaccurate depth values.

3. **Tracking**

    - Kernel 1: **halfSample** By subsampling the filtered depth image, a three-level image pyramid is formed. The tracking solutions from the pyramid's low-resolution images are used as guesses for higher resolutions. **depth2vertex**: translates each pixel in a new depth image into a three-dimensional point (vertex). As a consequence, a point cloud is created by this kernel.
    - Kernel 2: **vertex2normal** the normal vectors for each vertex of a point cloud are computed. Normals are used to measure the point-plane distances between two corresponding vertices of the synthetic point cloud and a new point cloud in the projective data association stage of the ICP algorithm.
    - Kernel 3: **track** in the synthetic and new point cloud, the correspondence between vertices is formed.
    - Kernel 4: **reduce** For the minimisation process, adds up all the distances (errors) between corresponding vertices of two point clouds. The final sum is computed on GPUs using a parallel tree-based reduction.
    - Kernel 5: **solve** TooN is used on the CPU to perform a singular value decomposition that solves a linear 6x6 method. To correct the new camera pose approximation, a 6-dimensional vector is generated.

4. **Integrate** The new point cloud is inserted into the 3D volume. The running average used in fusion is determined.

5. **Raycast** computes the point cloud and normals that lead to the actual camera location estimation.

6. **Render**

- Kernel 1: **renderDepth** color coding is used to visualize the depth map obtained from the sensor.
- Kernel 2: **renderTrack** Visualizes the monitoring performance. Different colors are correlated with each of the potential effects of the monitoring move with each pixel, e.g. 'right tracking',' pixel too far away',' false standard', etc.
- Kernel 3: **renderVolume** The 3D reconstruction is visualized from a fixed point of view (or a user specified viewpoint when in the GUI mode).

Kernels	Pipeline
acquire	Acquire
mm2meters	Preprocess
bilateralFilter	Preprocess
halfSample	Track
depth2vertex	Track
vertex2normal	Track
track	Track
reduce	Track
solve	Track
integrate	Integrate
raycast	Raycast
renderDepth	Rendering
renderTrack	Rendering
renderVolume	Rendering

Figure 2.5: Kernels Pipeline [5]

2.1.4 ICL-NUIM Dataset & Trajectory Error

SLAMBench can take as input video frames, instead of using a Kinect camera, or any camera sensor. We use ICL-NUIM datasets, containing camera poses and ground truth poses. One of the goals of ICL-NUIM datasets is to help in benchmarking SLAM systems, and that makes it a perfect match to use as algorithm's input. We convert given datasets to .raw files, using an already implemented script of SLAMBench named scene2raw. The trajectory error,

is the absolute distance between the projected trajectory and the ground truth. For visual SLAM systems, the Absolute Trajectory Error (ATE) is the most common error metric.

In this paper, we are using the living room trajectory 'lr kt2' loop from [16, 17] as input frames, which has a real run time of 30 seconds and 882 frames at 30Hz.

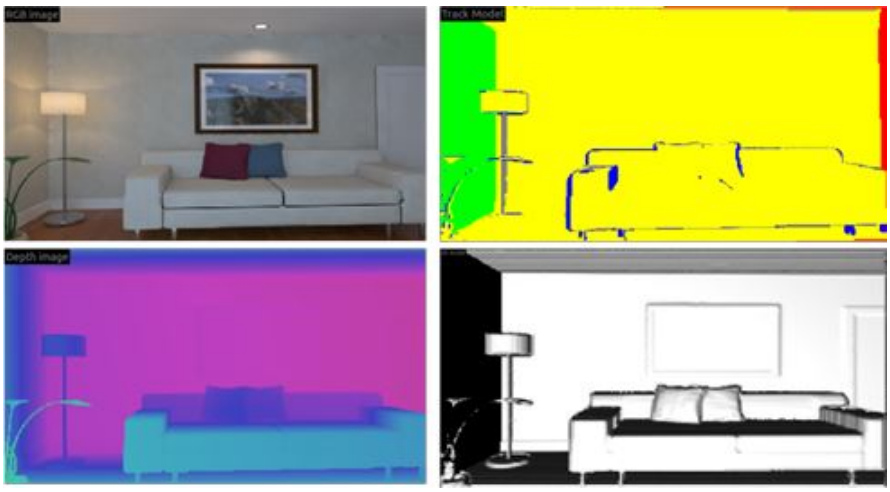

Figure 2.6: I/O from KinectFusion: the input RGB scene (top left), the input depth frame (bottom left), the tracking output (top right), and the 3D map reconstruction (bottom right) [6]

2.2 Device and Framework

As already stated before, in this thesis we focus on GPUs and how we can optimize the KinectFusion implementation. We use the OpenCL KinectFusion implementation of SLAMBench[5], and test our results on NVidia GeForce GTX 770 device.

OpenCL is a system for software writing that runs across heterogeneous platforms. Khronos Organization, a non-profit infrastructure association, manages it as an open standard.

A computer system in OpenCL [15] is made up of a variety of compute devices, such as CPUs or "accelerators" like GPUs, all of which are connected to a host processor, which is a central processor unit. On the way of writing, it is almost like a C programming language. The "kernels" are the functions performed on an OpenCL computer. Usually, a single computing system is made up of several compute units, each of which comprises several PEs. On all or all of the PEs in parallel, a single kernel execution will run.

Launched in May 2013, the NVIDIA GeForce GTX 770 desktop Graphics Processing Unit uses the Kepler architecture and is assembled using 28 nm technology. The card is clocked at 1.046 GHz and can be boosted as high as 1.085 GHz. It also has 1536 CUDA cores,

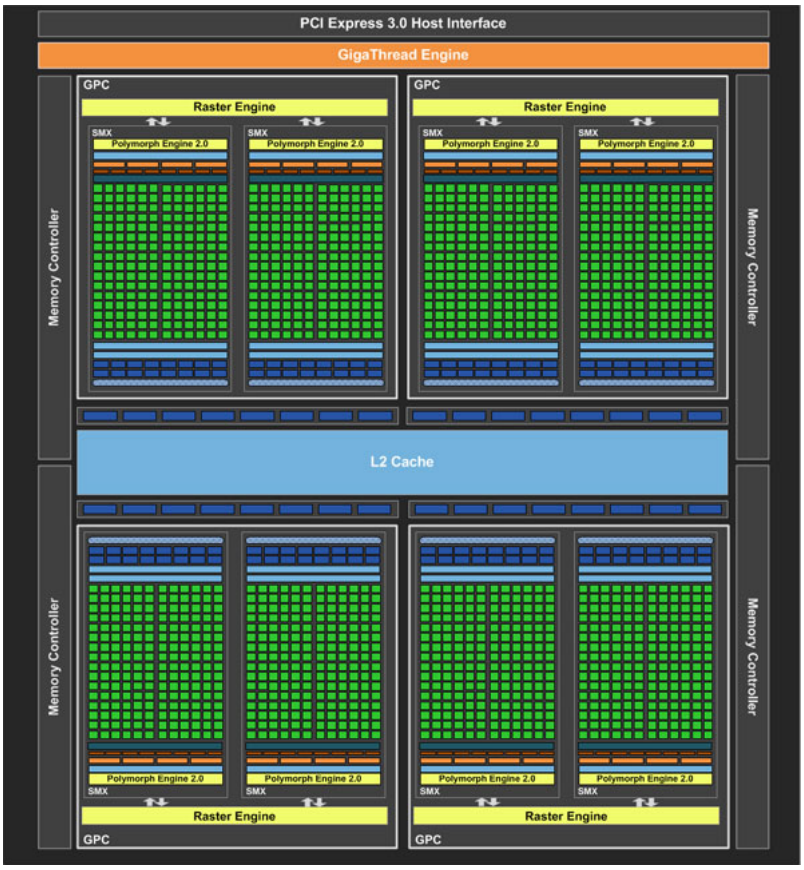

Figure 2.7: GK104 block diagram that entails the Kepler architecture [7]

128 texture modules, and 32 ROPs. The GeForce GTX 770 comes with 2 GB of GDDR5 memory. The memory uses a 256 bit interface and runs at 1.7525 GHz. As a result, the card's ram bandwidth is 224.32 GB/s. On the GPU, there is a PCI Express 3.0 interface. The average power consumption of this model is 195 Watts.

Frequency Base clock: 1046 MHz, GPU boost: 2.0, Boost clock: 1085 MHz

Memory specifications Memory size: 2048 MB, Memory type: GDDR5, Memory clock: 1752.5 MHz, Memory clock (effective): 7010 MHz, Memory interface width: 256-bit, Memory bandwidth: 224.32 GB/s

Cores CUDA: 3.5, CUDA cores: 1536

Performance Pixel fill rate: 34.72 Gigapixels/s, Texture fill rate: 138.88 Gigatexels/s, Single precision compute power: 3333.12 GFLOPS, Double precision compute power: 138.88 GFLOPS

2.3 Related Work

SLAM Systems add way too much overhead on commercial CPUs and mobile devices. It is only recently, that people have tried to accelarate SLAM algorithms through GPUs and FPGAS, achieving excessive performance.

Gkeka, Patras et al. [6] accelerated all kernels of KFusion algorithm from SLAMBench in an FPGA, without significant hurting the error. Gautier et al. [18, 19] proposed performance optimizations on FPGAs, on two different works. ORB-SLAM [11] is a sparse SLAM, which was implemented in an FPGA for better performance [20]. Abouzahir et al. [21] implemented FastSLAM2.0 [10] in both GPU and FPGA. Boikos and Bouganis [22] accelerated LSD-Slam on an FPGA achieving more than 60 fps. Lee et al. proposed a GPU-based SLAM system [23], and the same authors also worked on GPU-acceleration for Image feature-based real-time RGB-D 3D SLAM[24].

Chapter 3

KinectFusion Optimizations

In this chapter we will go through the optimizations we applied in the KinectFusion implementation of SLAMBench [5]. Different techniques work for different kernels in SLAMBench, as shown in [6], a work for accelerating the algorithm in FPGAs. Techniques that lead to significant performance speedup are mostly approximate, meaning that we allow the technique to hurt the error (ATE) in order to achieve better speedup. As a consequence, optimizations on individual kernels do not guarantee that all kernels can be combined to run the algorithm smoothly. For example, an optimization on Kernel 1, with low error affection, and an optimization on Kernel 2, with low error affection, when combined can lead to huge ATE or even code corruption.

We first optimized each kernel individually, and we present the optimizations. We then combine the kernels and the optimizations to produce the final optimized version of KinectFusion on OpenCL.

You can find the baseline kernels source code in the **appendices** 5. The optimizations described below are based on those lines of code. The default source code have been put on appendices, so it is easier to understand the optimizations while reading.

Optimizations are classified in two categories:

- **Precise** optimizations: The algorithm preserves precision and achieves better performance.

- **Approximate** optimizations: We sacrifice some of the accuracy to gain performance [25]

3.1 Baseline

KinectFusion is an algorithm that takes advantage of the GPU computation power, as it relies on computations. Before we start with the optimizations, we would like to mention the baseline performance.

We run the initial OpenCL implementation on GTX 770 with 8 compute units. I/O and Kernels took **5.21** seconds to complete all 882 frames, running at 203 frames per second. Before we continue we present the comparison between the GPU and CPU performance.

	NVIDIA GPU GTX 770	INTEL CPU i7-4820KU	Speed Comparison
ATE	18.118	18.113	-
acquisition $_{host}$	0.064	0.090	-
computation	4.352	28.696	x6.56
preprocessing	0.37	1.513	x4.89
tracking	1.905	5.359	x2.81
integration	0.515	7.625	x14.8
raycasting	1.556	14.496	x9.31
total	5.217	32.762	x6.27

Table 3.1: NVidia GPU GTX 770 vs Intel CPU i7-4820KU Performance Comparison

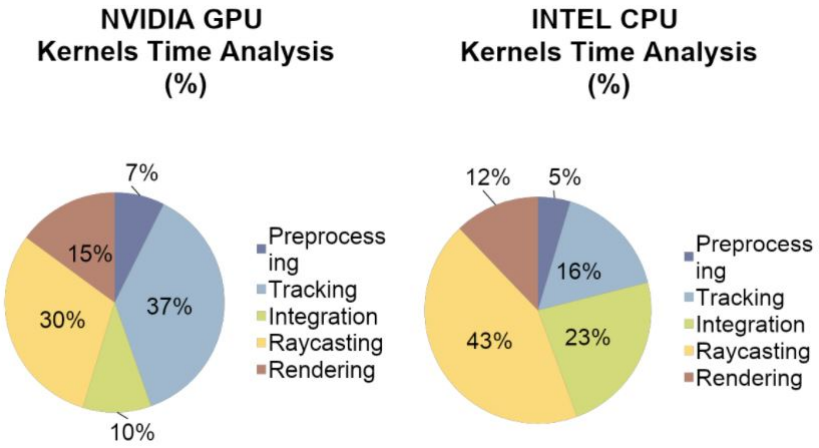

Figure 3.1: Where do kernels spent their time?

3.2 Bilateral Filter

Bilateral filter [26] is an edge-preserving blurring filter applied to the depth image. It reduces the effects of noise and invalid depth values. The kernel reads the input $320x240$ depth image, the $5x5$ filter, and writes back the new blurred image.

As stated in 3.1 prepossessing takes 0.37 seconds, which is already very good performance, compared to the same algorithm running on different other devices.

We decided that a precise technique, such as Input Depth Padding, is better to be implemented as an approximate technique. By making the input depth frame padded, kernels can eliminate boundary conditions and get rid of branch divergence, which can cause significant delays in our execution. Normally, when we pad an array on the host side we can set the exact values needed for the computations, so when the accelerator side accesses the padded part always take the correct value as if an if-else statement was in place. In our case, the kernel side is way too fast, and the CPU can not match the performance by copying the correct values to the correct positions. For this, we initialize the padded values to zero, barely affecting the error.

The baseline algorithm uses a $5x5filter$, taking into consideration all 25 neighbors of a single element. Although a smaller filter leads to a less smooth result, it results in fewer memory accesses and computations. We choose a $3x3filter$ for our convolution, using constant values. We take into consideration 9 neighbors of a single element. This causes the loop in the kernel to be unrolled 9 times.

We apply the following $3x3filter$.

gaussian[9] = 0.9394130111, 0.9692332149, 0.9394130111, 0.9692332149, 1, 0.9692332149, 0.9394130111, 0.9692332149, 0.9394130111

By applying a $3x3filter$, we are able to unroll the the nested loops 9 times, making each nearby element computations independent and more flexible in scheduling and executing.

We also remove the range filter, which is an exponential function. By doing this we eliminate the invocation of an exponent function in the filter.

Bilater Filter
Precise Optimizations
-
Approximate Optimizations
1. Input Depth Frame Padding (Partial) *2.* 3x3 Filter *3.* Loop Unroll *4.* Remove Range Filter

Table 3.2: Bilateral Filter Optimizations

3.3 Tracking

The Tracking kernel accesses the current image produced in the preprocessing stage. It pairs each pixel to its corresponding point in the 2D projection of the reconstructed model. Tracking Kernel is called multiple times for the three pyramid levels for each frame. As in the default execution, it is called for a maximum of 10, 5, and 4 iterations for level 0, level 1, and level 2 respectively.

We reduce the pyramid levels from the default three as stated above. This is an approximate technique. Pyramids are used for image detail enhancement. In our case, we are using low-resolution images, the depth resolution in $320x240$ unpadded. Skipping some of the processes will not dramatically affect image quality. We choose to reduce the levels to 1 instead of 3, and the maximum iterations to 2.

Loop perforation [27] is also an idea which implies great performance speedup. Loop perforation is a general term and refers to skipping some iterations by copying the result of an iteration to the skipped iterations. Although we do not have a loop in our implementation, as we have multiple kernels running for a single job, the idea remains the same. Instead of creating as many kernels as the elements, we create fewer kernels and copy results to element's neighbor. Although we tried different blocks and copies, it seems that trying to access different memory cells makes the overhead bigger than the gained performance, as the kernel execution time has fallen deeply after the first optimization described above. We evaluated different tiles both on the X-axis and Y-axis, horizontally and vertically. Starting from step 1 to 16, as after step equal to 16, the Trajectory Error would get deeply affected. We also tried

different block sizes, [2, 4] and [2, 8] but nothing seemed to match the performance of the first optimization. Bigger blocks would have given slightly better timing performance but would not perform the same regarding the ATE, and the trade-off did not worth the optimization.

In defense of the above, we also tried to copy the result of the skipped iterations on the host side, preventing any overhead that may occur due to memory banks. Although we did 72.000 fewer accesses with a step of 16 on the kernel side, the kernel run only 0.14 seconds faster, meaning that there was not such a problem. On the other hand, the overhead caused on the host side, for copying the values, was huge.

Tracking
Precise Optimizations
-
Approximate Optimizations
1. Reduce Pyramid Levels *2.* Reduce Iterations

Table 3.3: Tracking Optimizations

3.4 Integrate

The target of this kernel is to update a 3D voxel grid, consisting of $256x256x256voxels$, using the new pose of the agent obtained by tracking. Each voxel runs independently.

We first removed some unnecessary functions. We focused on square roots functions applied on equations and squaring functions but did not get a great performance speedup. We also tried to get rid of specific branches, to avoid divergence. There were different branches to be removed, some that would just jump an iteration because there was no need to perform computations (precise) and others that would affect the Trajectory Error (approximate). The trade-off between the Error and the speedup was not worth it, except the precise branch mentioned above, with a positive speedup close to zero.

The main optimization on this kernel again stands on the Loop Perforation idea. Although here the default kernel creates a 2D global workgroup, creating a thread for a single position on X, Y, the kernel also works on a third dimension (Z-axis) on voxels grid. On this specific kernel, every thread (representing an element) has to loop on the Z-axis. At this point, we can skip some iterations on the third axis without even copying the value as it does not affect the execution. We have to be careful to start from different points every time so that we avoid repetition and we will not be able to track frames. Also, we do not only have to jump a specific step every time, but also determine the new position and camera position. We chose to specify the starting point by the frame we currently at, and by specifying a constant step, every frame will go through different data. Take for example frame 0 and step 4, then frame 0 will set the volume for $z = 0, 4, 8$, frame 1 will set the volume for $z = 1, 5, 9$, and so on.

We tried different steps, but we will go through our evaluation in the next chapter.

As a last optimization, we thought of tiling. Creating a much smaller 2D grid, and copying the values to neighbors. At this point, the kernel performance was almost at zero, and we decided that it is pointless to continue with further approximate optimizations.

Integrate
Precise Optimizations
1. Remove branches
Approximate Optimizations
1. Remove functions
2. Loop Perforation
3. Tiling (-)

Table 3.4: Integrate Optimizations

3.5 Raycast

Raycast computes the point cloud and normals corresponding to the current estimate of the camera position. Raycasting accesses Truncated Signed Distance Function (TSDF) multiple times, for interpolation.

The ray traversal uses steps of variable size. It starts with larger step size and it becomes smaller and smaller as the ray approaches a surface or the edges of the 3D voxel grid. Instead, we use a constant step to achieve a deterministic schedule. Along with that, we modify the interpolation function, and instead of visiting 8 different positions on TSDF, we access two TSDF values, making less overall memory accesses.

We also applied raycast once every some number of pixels, similar to loop perforation. We try different steps and block sizes, and we present all of them in the evaluation chapter.

Finally, we chose to raycast at a lower frequency, every 2 frames instead of raycasting every single frame, the change in Error is really small on the particular Trajectory, that we could even consider this technique a precise technique, depending on the metric.

Raycast
Precise Optimizations
-
Approximate Optimizations
1. Larger Step *2.* Change Interpolation *3.* Perforation *4.* Lower Frequency

Table 3.5: Raycast Optimizations

3.6 Render

Rendering consists of 3 kernels, renderDepth, renderTrack and renderVolume. The target of this kernel is to take the data and visualize the depth map acquired from the sensor, visualize the result of tracking, and the 3D reconstruction from a fixed viewpoint.

All this information refers to GUI mode. In this thesis, we focus on computations and performance regarding the speedup of the KinectFusion algorithm.

For all the above, Render is not a kernel we have to take into consideration.

Chapter 4

Kernels Evaluation

In this chapter, we will go through the evaluation of each kernel individually and in combination. We will apply the optimizations proposed in 3. It is important to mention that some optimizations may work on a kernel when we optimize it individually, but when combining kernels, the circumstances change and we also need to change some parameters or even remove some optimizations. All the following total time results exclude the rendering kernel.

4.1 Individual Evaluation

4.1.1 Bilateral Filter

To observe a 1.3x speedup in Bilateral Filter optimized version we had to combine all the optimizations. Bilateral Filter ran with exceptional performance prior to the optimizations, so the potential was limited. Most of the speedup came from the reduction of the Gaussian filter.

	Baseline	Optimized
ATE	18.118m	19.086m
preprocessing	0.37s	0.29s
total	4.41s	4.38s

Table 4.1: Bilateral Filter Comparison between Baseline and Optimized

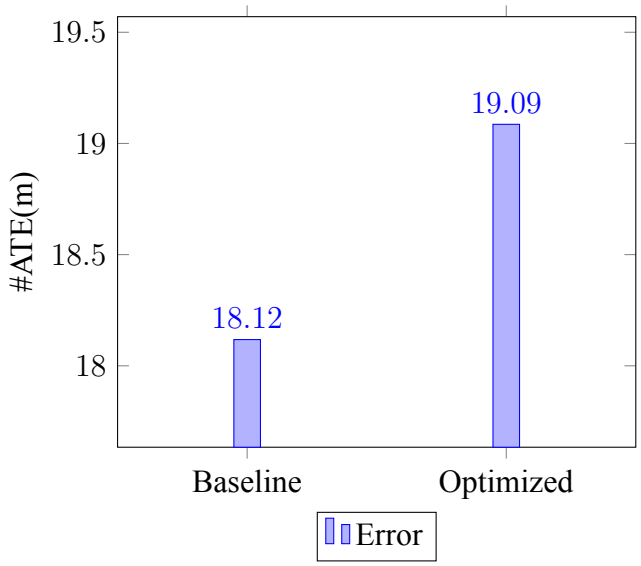

Figure 4.1: Bilateral Filter ATE Comparison between Baseline and Optimized

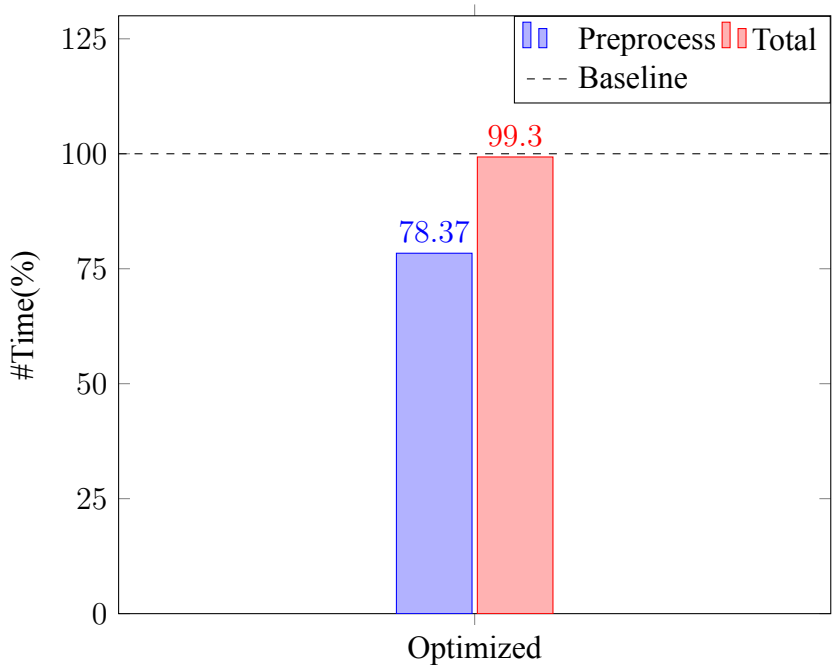

Figure 4.2: Bilateral Filter Performance Comparison between Baseline and Optimized shown in Percentages

4.1.2 Tracking

In this kernel, we applied two approximate optimizations, reducing the pyramid levels and iterations. As stated in previous chapter.

	Baseline	Optimized
ATE	18.118m	18.57m
tracking	1.91s	0.50s
total	4.41s	3.04s

Table 4.2: Tracking Comparison between Baseline and Optimized

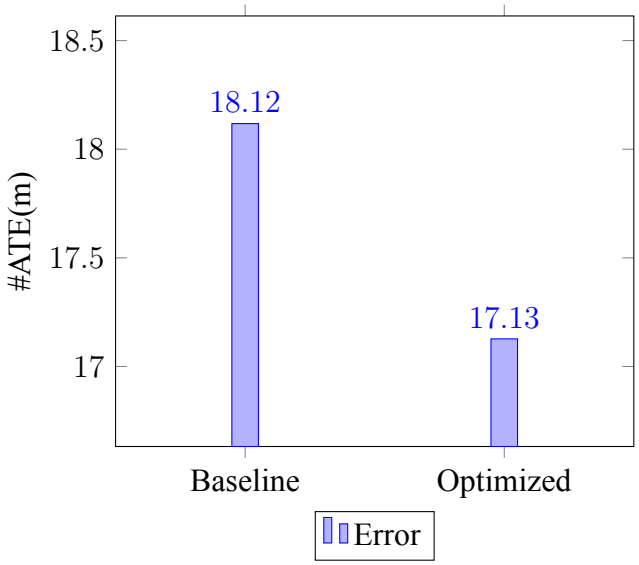

Figure 4.3: Tracking ATE Comparison between Baseline and Optimized

4.1.3 Integrate

In this kernel we applied three different optimizations. Removing functions and branches, and applying loop perforation technique.

	Baseline	Optimized
Remove Functions		
ATE	18.118m	18.88m
integration	0.52s	0.50s
Remove Branches		
ATE	18.88m	18.88m
integration	0.50s	0.48s

Table 4.3: INT_Opt1_Opt2

To perform loop perforation we needed to specify a loop step. In the following figures

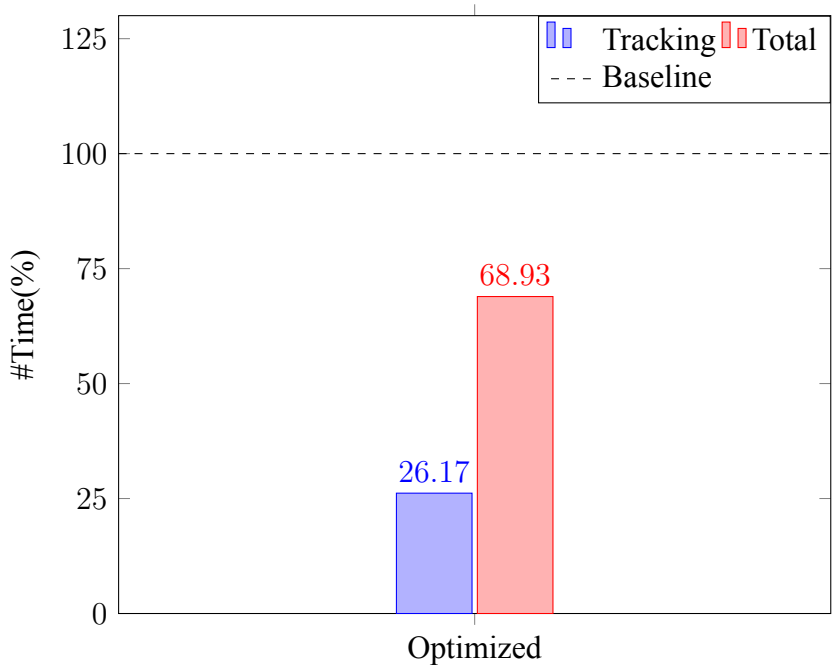

Figure 4.4: Tracking Performance Comparison between Baseline and Optimized shown in Percentages

4.5 4.6 we present the evaluation of different steps applied.

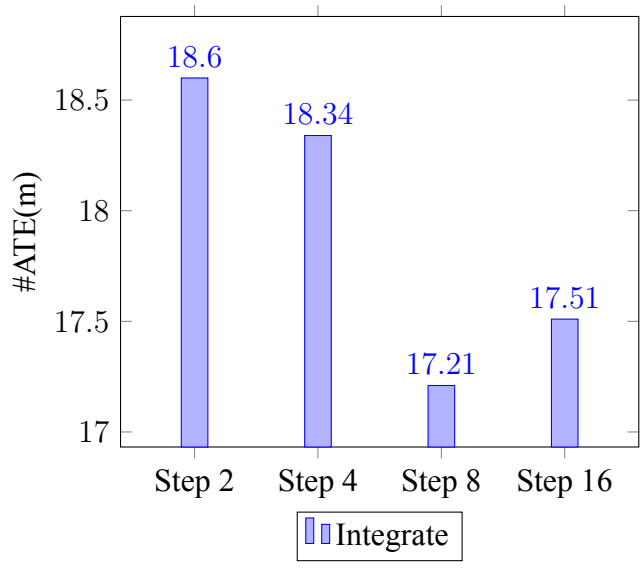

Figure 4.5: INT_Step ATE

Based on the information provided by figures, we applied step equal to 16 in the integration kernel.

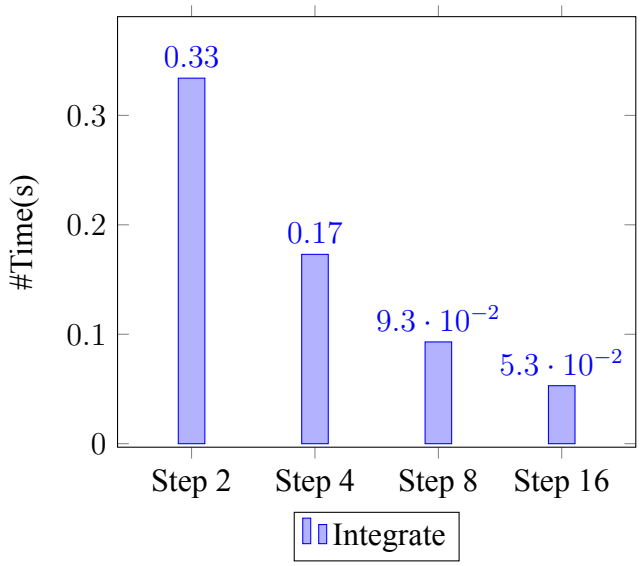

Figure 4.6: INT_Step Performance

	Baseline	Optimized
ATE	18.118m	18.35m
integrate	0.52s	0.053s
total	4.41s	4.07s

Table 4.4: Integrate Comparison between Baseline and Optimized

4.1.4 Raycast

As stated in 3 we applied 4 approximate techniques. First, we used constant value at LargerStep and we changed the interpolation function to access 2 values instead of 8, we applied those two techniques as one. We then performed the Loop Perforation technique and applied Raycast to lower frequency, specifically every two frames.

At Loop Perforation we applied a step on two axes, copying a block of elements, the evaluation is shown in 4.8 and 4.9. We decided to use a Block 1x2, which means we skip 1 on X-axis and 2 on Y-axis, and we copy the value to those positions.

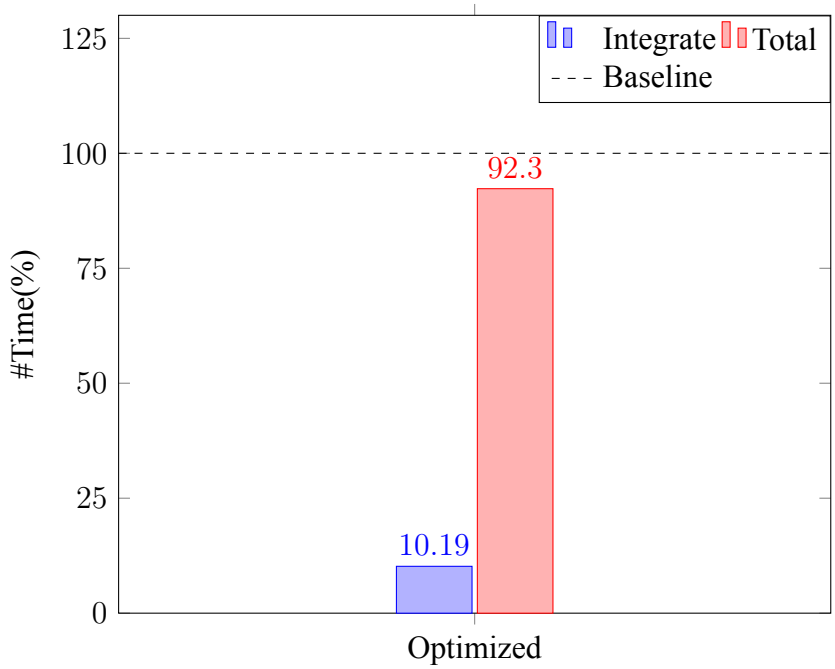

Figure 4.7: Integrate Performance Comparison between Baseline and Optimized shown in Percentages

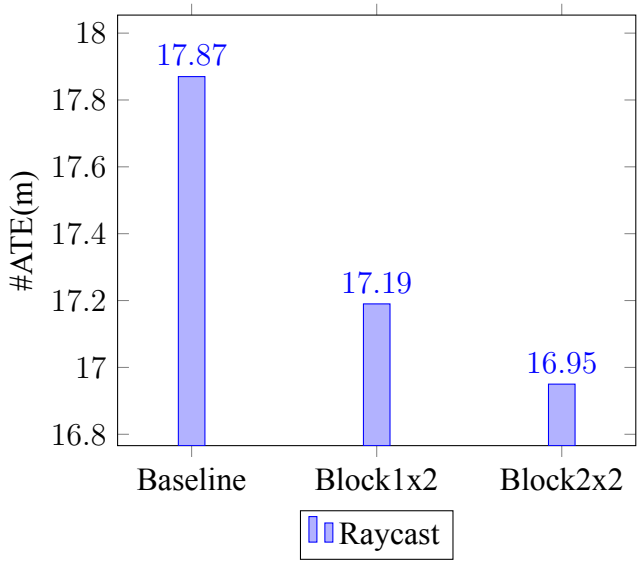

Figure 4.8: RAY_Step ATE

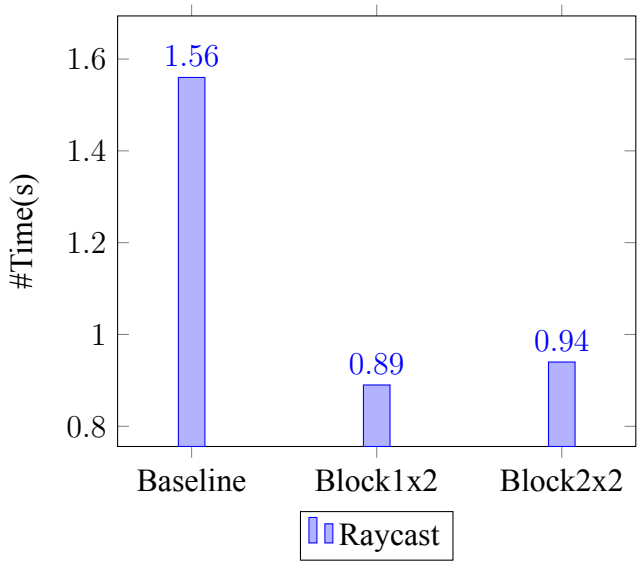

Figure 4.9: RAY_Step Performance

	Baseline	Optimized
Constant Step & Interpolation		
ATE	18.118m	17.87m
raycast	1.56s	1.15s
Perforation		
ATE	17.87m	16.66m
raycast	1.15s	0.67s
Lower Frequency		
ATE	17.87m	17.97m
raycast	0.67s	0.35s
total	4.41s	3.08s

Table 4.5: Raycast Comparison between Baseline and Optimized

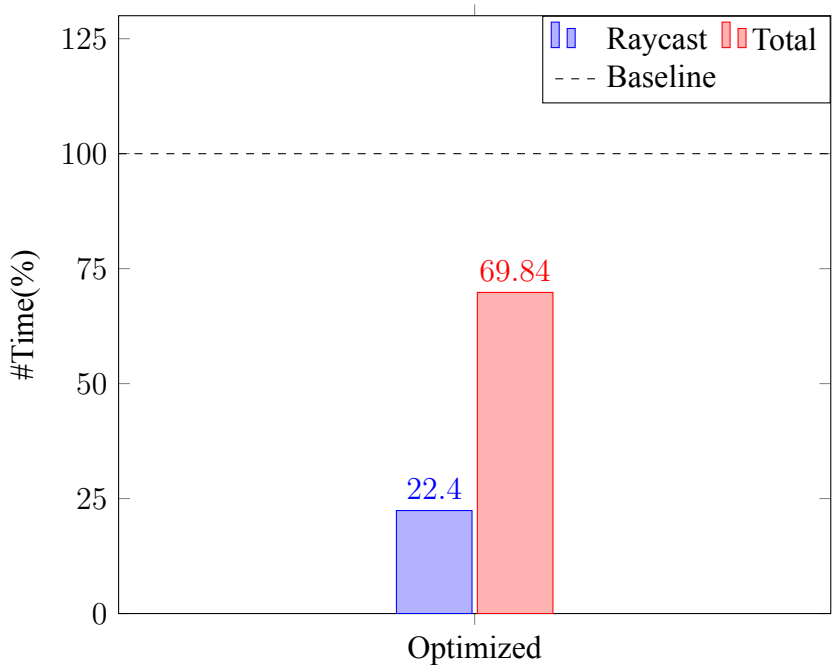

Figure 4.10: Raycast Performance Comparison between Baseline and Optimized shown in Percentages

4.2 Combining Kernels

In this section, we will combine our optimized kernels and apply the best possible parameters and techniques. Our target is to get an optimal solution out of the combination. We go step by step, adding one kernel at a time. We start by combining Integrate and Tracking kernel. We then continue with Preprocessing, Bilateral Filter relies on this overall process, and finally, we end up adding to our combination Raycast kernel. In the end, we provide our final evaluation on Living Room Trajectory of [16], where we managed to stay close to the baseline error, and we achieved exceptional timing performance.

4.2.1 Integrate and Tracking (IT)

Due to the algorithmic optimization of Tracking 3.3, the overall execution is not harmed at all. All frames are tracked as expected.

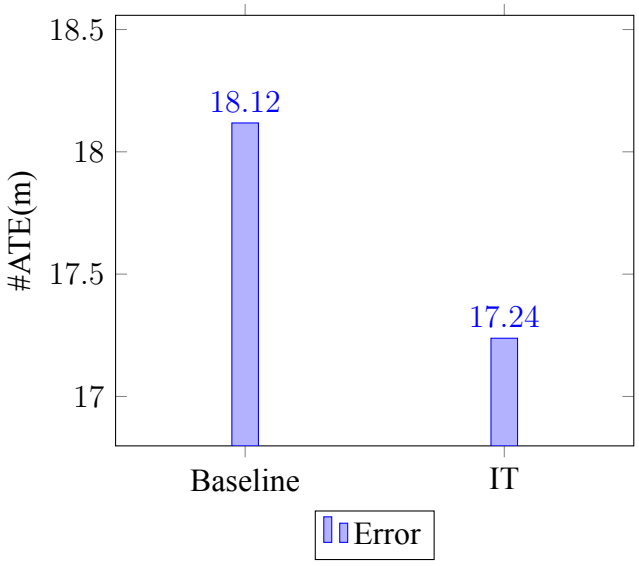

Figure 4.11: Integrate and Tracking ATE Comparison between Baseline and Optimized

4.2.2 Bilateral Filter and IT (BIT)

To combine the Bilateral Filter kernel with IT combination, we have to change the INT_Step, as Step equal to 16, with the new padded image, can not co-operate and produces an extremely large Trajectory Error. Once again we tested various steps and we concluded that for BIT combination we chose Step equal to 12 as shown in 4.14.

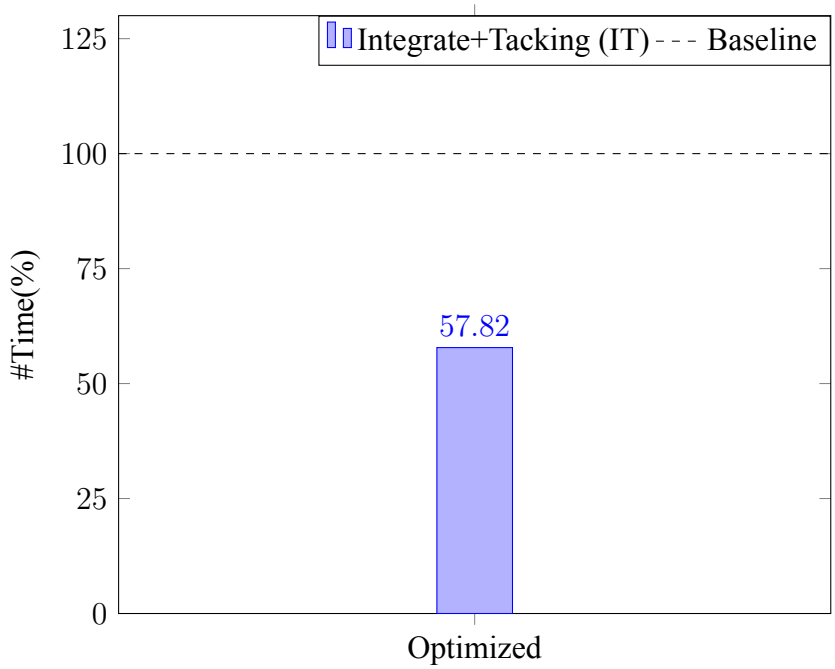

Figure 4.12: Integrate and Tracking Performance Comparison between Baseline and Optimized shown in Percentages

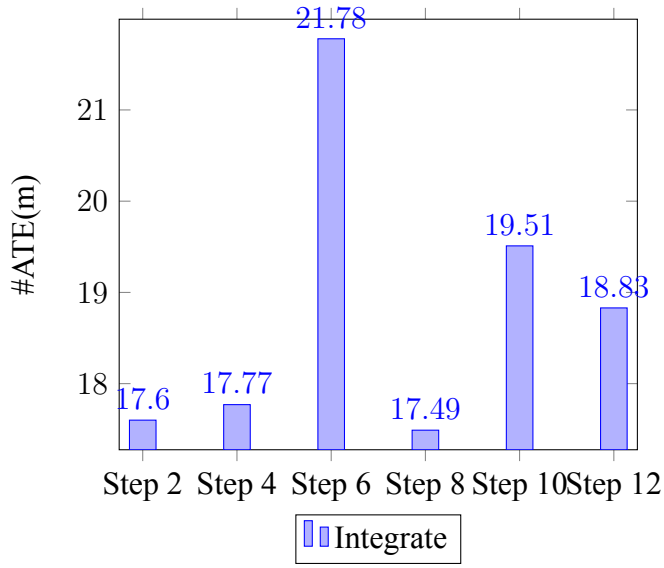

Figure 4.13: BIT_Step ATE

4.2.3 Raycast and BIT (RBIT)

Finally, combining Raycast Kernel we have once again to change the INT_Step 3.4. Although Step equal 12 still works fine, Step 8 almost matches the performance giving us a better trade-off according to Trajectory Error. For this, we chose Step equal to 8. Furthermore, at this combination, we had to remove RAY_ConStep 3.5 as it could not be combined

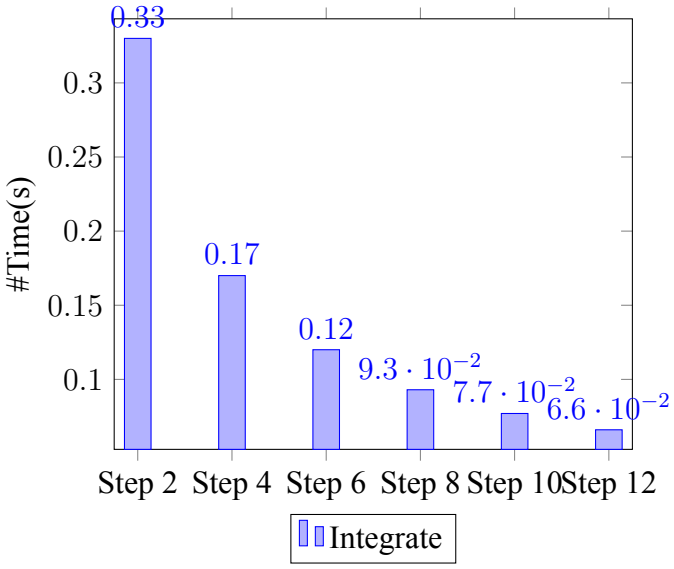

Figure 4.14: BIT_Step Performance

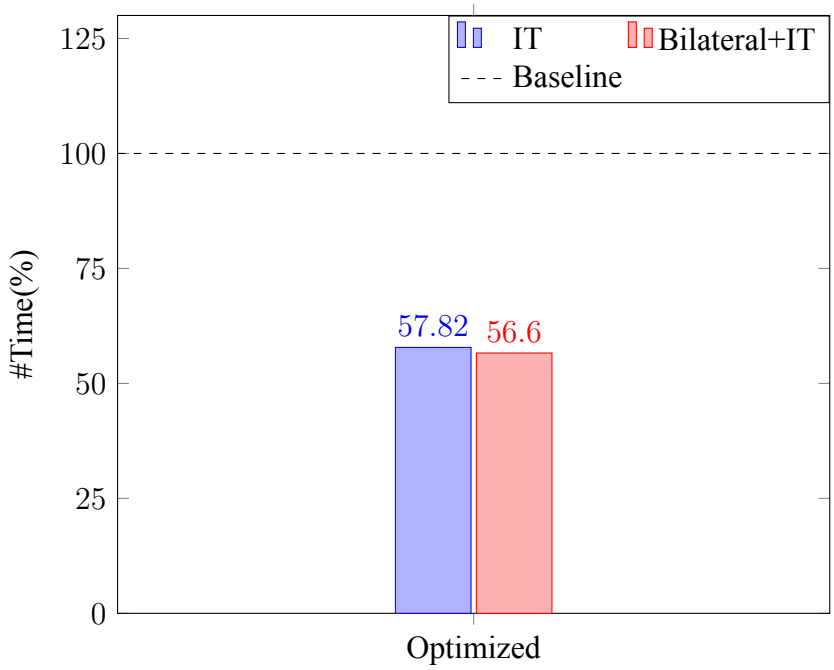

Figure 4.15: Bilateral Filter and IT Performance Comparison between Baseline and Optimized shown in Percentages

with other optimizations. As this is our final combination the results are shown in 4.3.

4.3 Overall Evaluation

We managed to achieve a 3.1x speedup to our optimized version. We run the KinectFusion algorithm on **634 frames per second**, 431 frames **more** than the baseline execution.

	Baseline	Final Optimized	Speed Comparison
ATE	18.2	19.38	-
preprocessing	0.37	0.30	x1.24
tracking	1.9	0.5	x3.8
integration	0.51	0.09	x5.67
raycasting	1.57	0.49	x3.2
total	4.41	1.39	x3.17

Table 4.6: Baseline vs Optimized Comparison Table

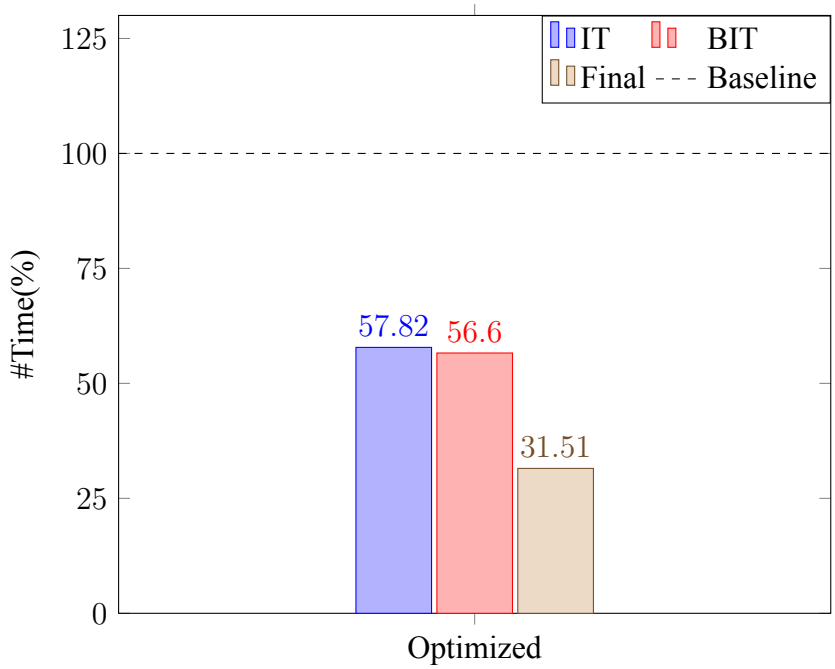

Figure 4.16: Optimized Performance shown in Percentages regarding the Baseline

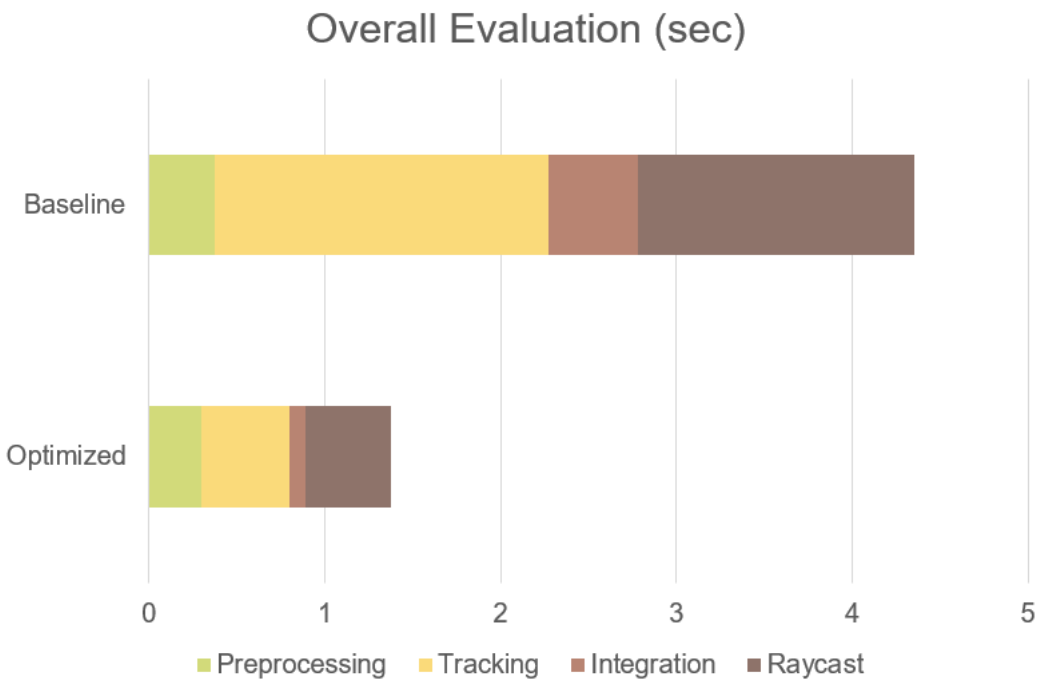

Figure 4.17: Final Evaluation Timing Performance

Chapter 5

Conclusions

In terms of time and precision, dense Visual SLAM algorithms must be fast and accurate. Since the agent relies on run-time computations, fast implementations are needed for powerful SLAM systems. We used numerous precise and approximate techniques to refine an OpenCL KinectFusion implementation in this thesis. We demonstrate that approximate techniques can achieve outstanding efficiency in GPU devices, but that they must be used with caution due to error sensitivity. At a 320x240 input depth frame resolution, our fastest implementation achieves 634 frames per second, which is 3.1 times faster than the baseline KinectFusion OpenCL implementation of SLAMBench.

The code of this thesis can be found at: https://github.com/pavlosaim

Bibliography

[1] Niklas Karlsson, Enrico Di Bernardo, Jim Ostrowski, Luis Goncalves, Paolo Pirjanian, and Mario E Munich. The vslam algorithm for robust localization and mapping. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 24–29. IEEE, 2005.

[2] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1052–1067, 2007.

[3] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*, pages 127–136. IEEE, 2011.

[4] Redhwan Jamiruddin, Ali Osman Sari, Jahanzaib Shabbir, and Tarique Anwer. Rgb-depth slam review. *arXiv preprint arXiv:1805.07696*, 2018.

[5] Luigi Nardi, Bruno Bodin, M Zeeshan Zia, John Mawer, Andy Nisbet, Paul HJ Kelly, Andrew J Davison, Mikel Luján, Michael FP O'Boyle, Graham Riley, et al. Introducing slambench, a performance and accuracy benchmarking methodology for slam. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5783–5790. IEEE, 2015.

[6] Maria Rafaela Gkeka, Alexandros Patras, Christos D Antonopoulos, Spyros Lalis, and Nikolaos Bellas. Fpga architectures for approximate dense slam computing.

[7] Palit geforce gtx 770 jetstream review - graphics architecture. `https://www.guru3d.com/articles-pages/palit-geforce-gtx-770-jetstream-review,4.html`. Accessed: 2021-06-15.

[8] Wikipedia. Simultaneous localization and mapping. https://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping. [Online;].

[9] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *European conference on computer vision*, pages 834–849. Springer, 2014.

[10] Michael Montemerlo, Sebastian Thrun, Daphne Koller, Ben Wegbreit, et al. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *IJCAI*, volume 3, pages 1151–1156, 2003.

[11] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[12] Alvaro Parra Bustos, Tat-Jun Chin, Anders Eriksson, and Ian Reid. Visual slam: Why bundle adjust? In *2019 International Conference on Robotics and Automation (ICRA)*, pages 2385–2391. IEEE, 2019.

[13] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al. Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568, 2011.

[14] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[15] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[16] A. Handa, T. Whelan, J.B. McDonald, and A.J. Davison. A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM. In *IEEE Intl. Conf. on Robotics and Automation, ICRA*, Hong Kong, China, May 2014.

[17] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *2014 IEEE international conference on Robotics and automation (ICRA)*, pages 1524–1531. IEEE, 2014.

[18] Quentin Gautier, Alric Althoff, and Ryan Kastner. Fpga architectures for real-time dense slam. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 83–90, 2019.

[19] Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner. Real-time 3d reconstruction for fpgas: A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl sdk. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 326–329, 2014.

[20] Weikang Fang, Yanjun Zhang, Bo Yu, and Shaoshan Liu. Fpga-based orb feature extraction for real-time visual slam. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 275–278, 2017.

[21] Mohamed Abouzahir, Abdelhafid Elouardi, Rachid Latif, Samir Bouaziz, and Abdelouahed Tajer. Embedding slam algorithms: Has it come of age? *Robotics and Autonomous Systems*, 100:14–26, 2018.

[22] Konstantinos Boikos and Christos-Savvas Bouganis. A scalable fpga-based architecture for depth estimation in slam. In *International Symposium on Applied Reconfigurable Computing*, pages 181–196. Springer, 2019.

[23] Donghwa Lee, Hyongjin Kim, and Hyun Myung. Gpu-based real-time rgb-d 3d slam. In *2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 46–48, 2012.

[24] Donghwa Lee, Hyongjin Kim, and Hyun Myung. Image feature-based real-time rgb-d 3d slam with gpu acceleration. *Journal of Institute of Control, Robotics and Systems*, 19(5):457–461, 2013.

[25] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)*, 48(4):1–33, 2016.

[26] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*, pages 839–846. IEEE, 1998.

[27] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134, 2011.

[28] Andreas Geiger, Julius Ziegler, and Christoph Stiller. Stereoscan: Dense 3d reconstruction in real-time. In *2011 IEEE intelligent vehicles symposium (IV)*, pages 963–968. Ieee, 2011.

APPENDICES

The following code represents the default unoptimized kfusion. For the GPU accelarated version visit my github page. https://github.com/pavlosaim/kfusion-opencl

```
__kernel void bilateralFilterKernel( __global float * out,
    const __global float * in,
    const __global float * gaussian,
    const float e_d,
    const int r ) {

  const uint2 pos = (uint2) (get_global_id(0),get_global_id(1));
  const uint2 size = (uint2) (get_global_size(0),get_global_size(1));

  const float center = in[pos.x + size.x * pos.y];

  if ( center == 0 ) {
    out[pos.x + size.x * pos.y] = 0;
    return;
  }

  float sum = 0.0f;
  float t = 0.0f;
  for(int i = -r; i <= r; ++i) {
    for(int j = -r; j <= r; ++j) {
      const uint2 curPos = (uint2)(clamp(pos.x + i, 0u, size.x-1),
          clamp(pos.y + j, 0u, size.y-1));
      const float curPix = in[curPos.x + curPos.y * size.x];
      if(curPix > 0) {
        const float mod = sq(curPix - center);
        const float factor = gaussian[i + r] * gaussian[j + r] *
            exp(-mod / (2 * e_d * e_d));
        t += factor * curPix;
        sum += factor;
      } else {
        //std::cerr << "ERROR BILATERAL " <<pos.x+i<< " "<<pos.y+j<< "
            " <<curPix<<" \n";
      }
    }
  }
  out[pos.x + size.x * pos.y] = t / sum;

}
```

Listing 5.1: Baseline OpenCL Kernel Code for Bilateral Filter

```
__kernel void trackKernel (
    __global TrackData * output,
    const uint2 outputSize,
    __global const float * inVertex,// float3
    const uint2 inVertexSize,
    __global const float * inNormal,// float3
    const uint2 inNormalSize,
    __global const float * refVertex,// float3
    const uint2 refVertexSize,
    __global const float * refNormal,// float3
    const uint2 refNormalSize,
    const Matrix4 Ttrack,
    const Matrix4 view,
    const float dist_threshold,
    const float normal_threshold
) {

  const uint2 pixel = (uint2)(get_global_id(0),get_global_id(1));

  if(pixel.x >= inVertexSize.x  pixel.y >= inVertexSize.y ) {return;}

  float3 inNormalPixel = vload3(pixel.x + inNormalSize.x *
     pixel.y,inNormal);

  if(inNormalPixel.x == INVALID ) {
    output[pixel.x + outputSize.x * pixel.y].result = -1;
    return;
  }

  float3 inVertexPixel = vload3(pixel.x + inVertexSize.x *
     pixel.y,inVertex);
  const float3 projectedVertex = Mat4TimeFloat3 (Ttrack ,
     inVertexPixel);
  const float3 projectedPos = Mat4TimeFloat3 ( view , projectedVertex);
  const float2 projPixel = (float2) ( projectedPos.x / projectedPos.z
     + 0.5f, projectedPos.y / projectedPos.z + 0.5f);
```

```
33
34  if(projPixel.x < 0  projPixel.x > refVertexSize.x-1  projPixel.y < 0
        projPixel.y > refVertexSize.y-1 ) {
35    output[pixel.x + outputSize.x * pixel.y].result = -2;
36    return;
37  }
38
39  const uint2 refPixel = (uint2) (projPixel.x, projPixel.y);
40  const float3 referenceNormal = vload3(refPixel.x + refNormalSize.x *
       refPixel.y,refNormal);
41
42  if(referenceNormal.x == INVALID) {
43    output[pixel.x + outputSize.x * pixel.y].result = -3;
44    return;
45  }
46
47  const float3 diff = vload3(refPixel.x + refVertexSize.x *
       refPixel.y,refVertex) - projectedVertex;
48  const float3 projectedNormal = myrotate(Ttrack, inNormalPixel);
49
50  if(length(diff) > dist_threshold ) {
51    output[pixel.x + outputSize.x * pixel.y].result = -4;
52    return;
53  }
54  if(dot(projectedNormal, referenceNormal) < normal_threshold) {
55    output[pixel.x + outputSize.x * pixel.y] .result = -5;
56    return;
57  }
58
59  output[pixel.x + outputSize.x * pixel.y].result = 1;
60  output[pixel.x + outputSize.x * pixel.y].error =
       dot(referenceNormal, diff);
61
62  vstore3(referenceNormal,0,(output[pixel.x + outputSize.x *
       pixel.y].J));
63  vstore3(cross(projectedVertex, referenceNormal),1,(output[pixel.x +
       outputSize.x * pixel.y].J));
64
65 }
```

Listing 5.2: Baseline OpenCL Kernel Code for Tracking

```
__kernel void integrateKernel (
    __global short2 * v_data,
    const uint3 v_size,
    const float3 v_dim,
    __global const float * depth,
    const uint2 depthSize,
    const Matrix4 invTrack,
    const Matrix4 K,
    const float mu,
    const float maxweight ,
    const float3 delta ,
    const float3 cameraDelta
) {

  Volume vol; vol.data = v_data; vol.size = v_size; vol.dim = v_dim;

  uint3 pix = (uint3) (get_global_id(0),get_global_id(1),0);
  const int sizex = get_global_size(0);

  float3 pos = Mat4TimeFloat3 (invTrack , posVolume(vol,pix));
  float3 cameraX = Mat4TimeFloat3 ( K , pos);

  for(pix.z = 0; pix.z < vol.size.z; ++pix.z, pos += delta, cameraX +=
      cameraDelta) {
    if(pos.z < 0.0001f) // some near plane constraint
    continue;
    const float2 pixel = (float2) (cameraX.x/cameraX.z + 0.5f,
        cameraX.y/cameraX.z + 0.5f);

    if(pixel.x < 0  pixel.x > depthSize.x-1  pixel.y < 0  pixel.y >
        depthSize.y-1)
    continue;
    const uint2 px = (uint2)(pixel.x, pixel.y);
    float depthpx = depth[px.x + depthSize.x * px.y];

    if(depthpx == 0) continue;
```

```
34    const float diff = ((depthpx) - cameraX.z) *
          sqrt(1+sq(pos.x/pos.z) + sq(pos.y/pos.z));
35
36    if(diff > -mu) {
37      const float sdf = fmin(1.f, diff/mu);
38      float2 data = getVolume(vol,pix);
39      data.x = clamp((data.y*data.x + sdf)/(data.y + 1), -1.f, 1.f);
40      data.y = fmin(data.y+1, maxweight);
41      setVolume(vol,pix, data);
42    }
43  }
44
45 }
```

Listing 5.3: Baseline OpenCL Kernel Code for Intergate

```
1  __kernel void raycastKernel( __global float * pos3D, //float3
2    __global float * normal,//float3
3    __global short2 * v_data,
4    const uint3 v_size,
5    const float3 v_dim,
6    const Matrix4 view,
7    const float nearPlane,
8    const float farPlane,
9    const float step,
10   const float largestep ) {
11
12  const Volume volume = {v_size, v_dim,v_data};
13
14  const uint2 pos = (uint2) (get_global_id(0),get_global_id(1));
15  const int sizex = get_global_size(0);
16
17  const float4 hit = raycast( volume, pos, view, nearPlane, farPlane,
        step, largestep );
18  const float3 test = as_float3(hit);
19
20  if(hit.w > 0.0f ) {
21    vstore3(test,pos.x + sizex * pos.y,pos3D);
22    float3 surfNorm = grad(test,volume);
```

```
23      if(length(surfNorm) == 0) {
24        //float3 n = (INVALID,0,0);//vload3(pos.x + sizex * pos.y,normal);
25        //n.x=INVALID;
26        vstore3((float3)(INVALID,INVALID,INVALID),pos.x + sizex *
              pos.y,normal);
27      } else {
28        vstore3(normalize(surfNorm),pos.x + sizex * pos.y,normal);
29      }
30    } else {
31      vstore3((float3)(0),pos.x + sizex * pos.y,pos3D);
32      vstore3((float3)(INVALID, INVALID, INVALID),pos.x + sizex *
            pos.y,normal);
33    }
34  }
```

Listing 5.4: Baseline OpenCL Kernel Code for Raycast