Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017

Project Title:	**Evaluation of Hardware-accelerated SLAM Algorithms**
Student:	**Arshan Shafiei**
CID:	**00826335**
Course:	**EIE4**
Project Supervisor:	**Dr. Christos-Savvas Bouganis**
Second Marker:	**Prof. Peter Cheung**

ABSTRACT

Simultaneous localization and mapping (SLAM) allows mobile robots to build and update the map of an unknown environment they are moving in, while tracking the position of the robot itself. One of the established SLAM algorithms is KinectFusion. It builds a rapid detailed 3D reconstruction of the scene taken by a Kinect camera in real time, uses the depth data collected by Kinect sensors and fuses them into a single 3D model. The open-source SLAMBench, used for benchmarking SLAM algorithms, provides an OpenCL implementation of KinectFusion. The Intel FPGA SDK for OpenCL enables programmers to run OpenCL code on FPGAs, abstracting away the difficulty of low-level hardware SLAM design.

As part of this project, the performance of the KinectFusion algorithm running on an FPGA device has been evaluated against a modern CPU and GPU within SLAMBench framework. Several optimization strategies have been applied to improve the performance of the algorithm on an FPGA device, while analyzing the execution time-area trade-offs. The approach has mainly been loop-pipelining the kernels using a single-work-item approach, introducing Intel FPGA SDK for OpenCL channels and combining the kernels if applicable. The results show that the targeted FPGA can be used as a suitable hardware accelerator for parts of the SLAM algorithm, achieving a frame rate of 8 FPS when combined with OpenMP acceleration on CPU. By applying FPGA-specific optimizations, the targeted FPGA outperforms multi-core CPU in terms of execution time, while consuming less power than the GPU.

Acknowledgement

I would like to express my sincere gratitude to Dr.Bouganis for supervising me throughout this challenging project. The vision he gave to me in designing software and how it maps to the underlying hardware is an invaluable skill I received from him in this field.

I am very grateful to have Kouris Alexandros and Stylianos Venieris advising me on different questions I had and helping me with issues I faced with the servers. Moreover, some special thanks go to Dr. James Davis, who helped me a lot with Intel FPGA SDK for OpenCL compilations issues.

Final thanks go to my family, who live far away from me but their unconditional love and support are always with me, especially during this four years of MEng course at Imperial College London.

Table of Contents

1 Introduction

1.1 Motivation

Computer vision algorithms have seen major advancements in recent years. These algorithms use image processing techniques to enable functionalities like object recognition, image restoration, scene reconstruction and face detection, which have several useful practical applications in real world. The performance of these algorithms is crucial when choosing what platform best meets the requirements of the design. Therefore, analysing the behaviour of these algorithms running on different platforms in terms of performance, accuracy, area usage and energy consumption is a main field of research in this field.

Simultaneous localisation and Mapping (SLAM) is one of the computer vision applications enabling mobile robots like Unmanned Aerial Vehicles (UAV) and self-driving cars to navigate through an unknown environment while keeping track of their position. Once the robot has the measurements from the sensors, it uses the relevant algorithm to estimate the robot pose and reconstruct the environment. It is the fundamental concept in autonomous robots with several applications in real world. Examples includes underwater robot target reacquisition mission [1], precision agriculture maps [2] and more recently found in modern systems like Dyson 360 eye, Google Tango project and Microsoft HoloLens [3]. SLAM algorithms are computationally expensive algorithms which require a lot of power and their performance varies much depending on what sensor, platform, robot dynamic and environment they are used in.

To remove the inconsistency involved in evaluating the performance of SLAM algorithms on different hardware platforms due to the environmental and sensors factors introduced above, SLAMBench [4] provides a benchmark for fair evaluation of a SLAM system. The framework provides an implementation of a SLAM algorithm called KinectFusion using OpenCL. Input to SLAMBench can be standard formats(OpenNI), directly from the camera (RGB-D camera like Microsoft Kinect), raw format or ICL-NUIM dataset which is a set of pre-recorded scenes allowing a reliable investigation of trade-offs between accuracy and performance of the targeted SLAM application.

Furthermore, due to the ongoing demand of performing computer vision applications and in particular SLAM in real time on an embedded space, parallelization plays a crucial role in development of their algorithms. This has resulted in programmers to use high level synthesis to program hardware in order to fully exploit the highly paralleled and low power capabilities of the reconfigurable hardware platforms, especially when it comes to complex probabilistic

software model like SLAM. This abstraction, could potentially solve the power and performance issues in an embedded space for solutions of SLAM applications.

Therefore, this project has implemented and evaluated the performance of the KinectFusion algorithm as part of the SLAMBench framework, on an FPGA device, using OpenCL as the high-level synthesis language. The project has applied several optimization techniques to optimize the algorithm on an FPGA device and assessed using different metrics, the suitability of running the KinectFusion algorithm on an FPGA device, by comparing the metrics with a GPU, CPU and multi core CPU.

1.2 Related Work

SLAM was first introduced by Smith and Cheeseman in the 1980's [5] and have received considerable attention since its introduction. The problem definition of a SLAM can be mathematically represented using the following: Given a series of sensors readings r_t taken over discrete time steps t, the SLAM algorithm tries to find the location of the robot x_t and the map of the surroundings m_t by computing the probability [6]:

$$P(x_t, m_t \mid r_{1:t})$$

One the most common approaches of solving SLAM is using Extended Kalman Filter (EKF). "By maintaining a covariance matrix which encompasses all landmarks, this method allows the EKF to develop pose and landmark estimates incrementally" [7]. Another common approach to solve SLAM is to use Particle Filters, where posterior distribution of robot poses and maps are approximated [7]. In particular, Rao-Blackwellised Particle Filter (RBPF) use a particle filters where each hold an individual map of the environment [8]. SLAM algorithms use mostly one of the above techniques to track, map and apply global optimisations steps [9] to enable SLAM with different sensors.

Various sensors can be used to facilitate SLAM. Most common examples include using laser rangefinders [10] to Sonar sensors [11], ultrasonic sensors, vision sensors [12] and cameras, with pros and cons associated to each [5]. Laser sensors can be mostly expensive and mostly use Extended Kalman Filter (EKF) for SLAM solutions. On the other hand, cameras tend to use Rao-Blackwellised Particle Filter (RBPF) to obtain the robot pose and reconstruct the environment, which result in large amount of processing information [5]. Whatever method and senor used, the embedded SLAM field suffers from a huge power and heavy computational requirements.

This can be a real issue in SLAM embedded applications. Often algorithms have tended to reduce the complexity of problem by assuming parameters about the environment, evaluating them in a particular scenario rather than in general environment to reduce the computational and power cost. For instance, [13] presents localization using an upward looking camera under strict illumination changes. Another approach has been to send the computationally expensive SLAM algorithm to base stations for processing, featuring a client-server and cloud model [14].While this method offloads the heavy computation to robust processors, it results in loss of data due to the high bandwidth requirements of SLAM algorithms with heavy power consumptions.

Having the afore-mentioned constraints on power and performance in place, mobile robots that do the SLAM processing on-board need to use light-weight software implementations on lower power platforms [9]. This results in a trade-off between the accuracy and performance of their algorithms; nevertheless, recent applications in robotics require both parameters to be high. Therefore, a possible solution to this issue could be using reconfigurable hardware platforms. In addition to that, high level synthesis languages (HLS) have resulted programmers to focus on the software design without much knowledge of underlying hardware configurations, resulting in computationally expensive real-time algorithms to move to the embedded space on reconfigurable hardware platforms and FPGAs.

Recent advancement in FPGAs have made them faster in computation, more flexible with regards to on chip-memories, logic elements and routing delays with less power consumption. FPGAs can be reconfigured to perform different functionalities with low cost and low risk. They consist of many different building blocks like lookup tables (LUTs), arithmetic hardware units, registers and recently floating point units. They offer significant advantages over ASICs, in that if any bugs are found on an FPGA device, rather than going through all the process of prototyping and manufacturing again, they can just be reprogrammed by the designer again and let the compiler program the hardware data paths for that functionality. Furthermore, development of FPGA SoCs that combine CPUs and FPGAs, have enabled more flexibility in moving algorithms to the embedded world.

In particular, due to the fact that SLAM algorithms require high bandwidth with low latency characteristics, FPGAs offer unique parallel processing techniques, and have successfully been used on many occasions to accelerate computer vision algorithms achieving high processing frame rate in real time with lower power consumptions [15]. For example, the real time Stereo vision system implementation on FPGAs designed by Jin et al. [16] resulted in 230 times speed up compared to the conventional software implementation. Another good example is FPGA-based feature detection application done by Bouris et al. [17] which employs SURF algorithm and supported processing of standard video (640x 480 pixels) with 56 frames per second,

performing 8 times speedup compared to an Intel CPU. In the latter, while the targeted FPGA consumes only 20W, a GPU consumes 200W executing the same algorithm [17].

An example of real time SLAM system offloaded to FPGAs in the past is [15] where a combination of FPGA and mobile CPU resulted in dense images with 752x480 pixel resolutions being processed at 60 frames per second with less than 2 milliseconds latency. Another example is the work done by Grigorios Mingas et al. [18] which concluded an FPGA implementation of SGM-SLAM algorithm perform 14.83 times faster compared to the software algorithm without significant loss in accuracy. Moreover, [19] presents an efficient matrix multiplication hardware accelerator which is used in a wider SLAM algorithm based on Extended Kalman Filter (EKF). Their FPGA optimization allowed EKF block throughput to rise from 6 to 44Hz.

With this in mind, FPGAs can be considered as possible hardware accelerators in SLAM applications. Therefore, this project has contributed by implementing and optimizing the FPGA platform to the SLAMBench framework using OpenCL. The project has analysed several optimization strategies and discusses what CPU-FPGA architecture design would result in the best performance in the benchmark with regards to execution time, area and accuracy.

1.3 Report structure

Below is a brief description of this report's structure:

- Section 1: Overview of SLAM and why FPGAs can be a good target for SLAM applications
- Section 2: Introducing OpenCL and how they map to FPGAs including overview of KinectFusion algorithm
- Section 3: Introducing techniques to optimize OpenCL codes on FPGAs and what methodology is used
- Section 4: performance evaluation of every kernel in the algorithm on several platforms including FPGA, CPUs and GPUs
- Section 5: Discussion of results and how kernels can be grouped together to result in the best performance on the FPGA device.
- Chapter 6: Evaluating the whole application, by comparing with other tested platforms in SLAMBench
- Chapter 7: Project conclusions and how it can be taken further ahead in future work

2 Requirements Capture

The project is intended to use one of the high-level synthesis languages (OpenCL) on FPGAs. High level synthesis languages (HLS) have given great advantages to system designers, abstracting away the circuit, logic and functional level design mapping the algorithm to hardware under the hook. While this is beneficial for complex designs and results in more flexibility, it is still useful to have a picture of the low-level architecture and how this mapping is done from high level point of view.

2.1 Software OpenCL

"OpenCL (Open Computing Language) is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms" [20]. It allows programmers to access the compute power of CPUs, GPUs and other processors from a single source code [21]. It has C-based syntax and has recently been introduced as HLS to FPGAs by Altera. This section will provide the key aspects of OpenCL applications that form the foundation of the design in this project.

2.1.1 OpenCL mechanism

An OpenCL application is consisted of a host connected to one or more OpenCL devices. Thus, the whole application is implemented in two parts: the host code and the kernel code (.cl file). The host code (typically a C++ file) runs in sequential on the host CPU until the kernel command is submitted. Soon after, the kernel code is queued on the OpenCL device (GPU, FPGA, multi-core CPUs, DSPs etc.) and the parallel computation starts on the kernel device, returning to the host code after the kernel computation is done. There are available APIs, allowing the host code to interface with the device and create a context to communicate with it. Clearly, the purpose of doing this memory transfer from host to the kernel device in the application is to increase the performance (execution time) of the targeted algorithm by exploiting parallelism capabilities of the OpenCL device.
In our case, the OpenCL device is an FPGA. While the architecture of how OpenCL model is specifically executed on an FPGAs is described later in the report (section 2.2.3), all OpenCL devices have the same model as shown in figure 2.1 [21].

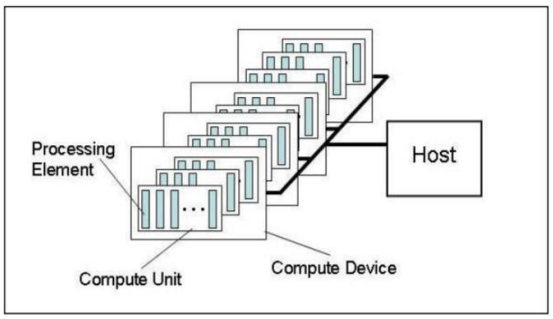


Figure 2.1: Host-kernel model of OpenCL application

Every OpenCL device is subdivided to compute units and further in to processing elements. A complex program that needs to be speed up typically consists of an iteration space, which defines how many number of times an algorithm (the kernel instance) should be executed.

2.1.2 Iteration space

The iteration space is called NDRange in OpenCL applications. NDRange is the index problem size, usually represented as (X, Y, Z) where each is the size of each of the dimensions in the overall iteration space. A good way of thinking about an NDRange representation is how for-loops are executed. A 1-dimensional problem size is basically a single for-loop represented as X in NDRange. Similarly, two nested for loops are represented as (X, Y) in NDRange.

When the host code issues commands to execute the kernel on the device, an instance of the kernel is executed, which is called a "work-item" in the iteration space. Each work-item is associated with a unique global ID in the entire iteration space. Work items are grouped together to create work-groups as shown in Figure 2.2 [21]. A work-item (kernel) is also associated with a unique local ID within its local work-group. Similarly, work-groups are assigned IDs within the entire iteration space. According to the specification, "When launching the kernel for execution, the host code defines the grid dimensions, or the global work size. The host code can also define the partitioning to work-groups, or leave it to the implementation [22].

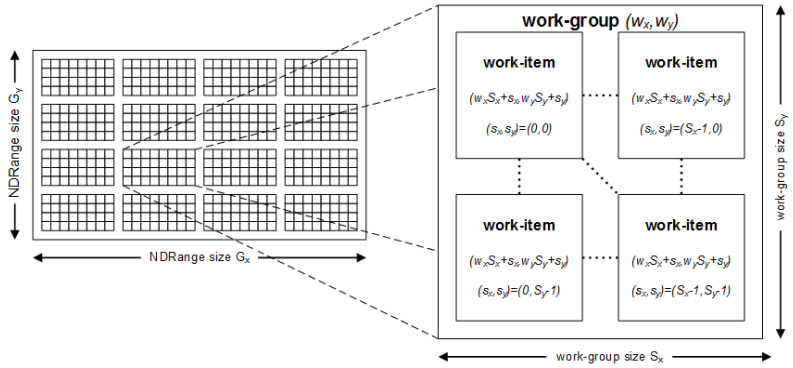


Figure 2.2: NDRange index size showing the entire grid iteration space and how it divides to work-groups and work-items

2.1.3 Memory hierarchy

In general, best memory accesses are those which never happen. The memory side, in particular, for FPGA devices, is an important aspect of the application and plays a crucial role in performance of the algorithm running on the device. Figure 2.3 illustrates the memory hierarchy of an OpenCL application.

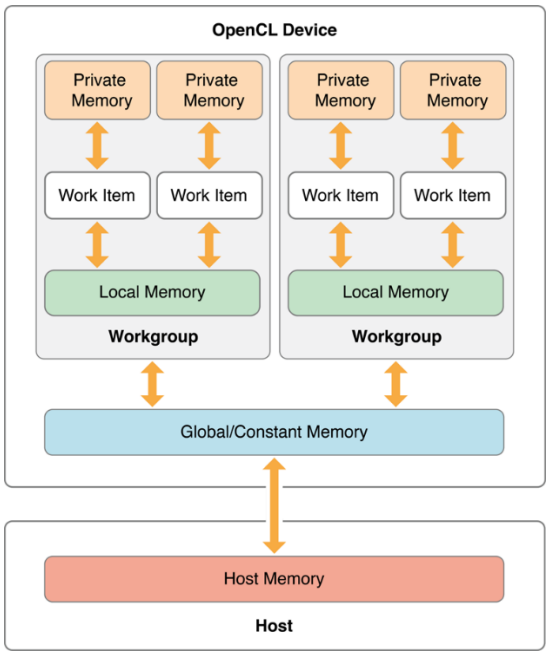


Figure 2.3 [23]: Memory hierarchy of an OpenCL applications

As can be seen, the memory hierarchy of OpenCL device consists of Global memory, constant memory, local memory and private memory. The important concepts here are:

- The global memory is accessible to all work-items in any work group. The host communicates with the OpenCL device through a Peripheral Component Interconnect Express (PCIe) bus.
- Constant memory is a section within the global memory that stays constant throughout the kernel execution. The host explicitly declares constant variables as arguments to the kernel, which gets placed in constant memory.
- Local memory is a section only accessible to work-items within a local work group.
- Private memory is a section only available to a specific work item and not any others other work items.
- Similar to the memory hierarchy in CPU (the larger the memory, the slower the access time), accessing global memory takes much more clock cycles than working with local memories within each work group. The same also applied to local and private memory comparison.

2.2 Hardware OpenCL

Accelerating code using OpenCL significantly lies on what hardware it runs on. although OpenCL offers kernel portability from CPUs, GPUs, and FPGAs, it does not guarantee that a kernel will achieve a maximum performance on a given device [24]. This is because different platforms have totally different underlying architectures and require different optimization techniques to be applied to achieve optimal performance. This section describes briefly how OpenCL acceleration is achieved by mapping the described model to relevant hardware platforms used throughout this project.

2.2.1 GPU OpenCL

GPUs have a unique architecture that make OpenCL harness their great processing power in high performance computing applications. GPUs can handle thousands of threads (work-items) simultaneously with their high number of cores. Combining the OpenCL mechanism with the index space, the global NDrange iteration space is mapped to the GPU device, where the work-groups are mapped to compute units and work-items (one instance of the kernel) are mapped to the processing elements as shown below (Figure 2.4) [25].

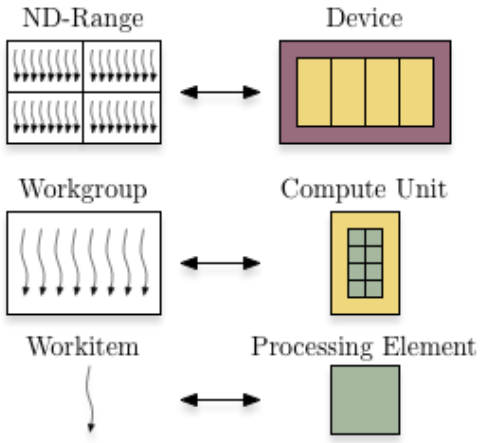


Figure 2.4: Mapping OpenCL to a GPU device

2.2.2 CPU OpenCL

Multicore CPUs can be used with OpenCL to increase the performance of OpenCL applications. CPUs have fewer processing elements but more memory than GPUs [23]. In fact, because modern CPUs have more vector units, multi-core CPUs can sometimes perform better than GPUs depending on input size [26]. OpenCL on CPUs leverages the computing power of multi-core CPUs, by executing the application parallel on the cores.

2.2.3 FPGA OpenCL

The concept of OpenCL on FPGAs is a new concept. It was introduced by Altera (later acquired by Intel) in 2011 exploiting highly parallel architecture of FPGAs with the OpenCL model. The tool which enables this is called the "Intel FPGA Software Development Kit (SDK) for OpenCL, which builds the custom FPGA hardware accelerator, adds interface IPs, builds interconnect logic and generates the bitstream required to program the FPGA from the high-level OpenCL kernel code" [27]. Unlike GPUs that load and build the kernel code at run-time, FPGAs OpenCL model requires kernel compilation to do the RTL synthesis.

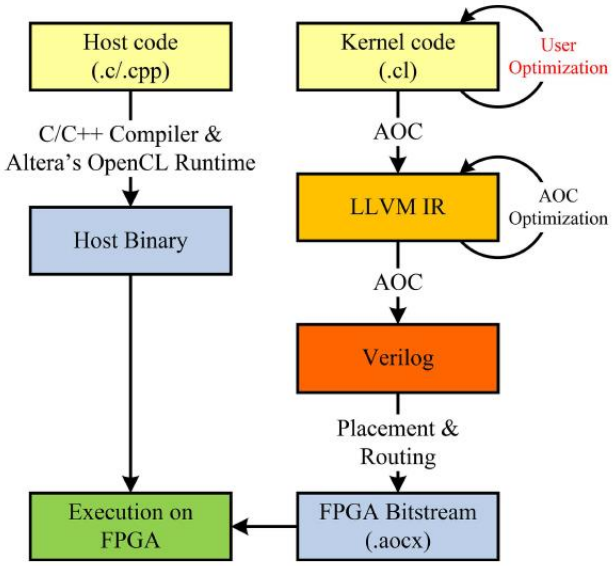


Figure 2.5: FPGA OpenCL model [28].

2.2.3.1 Kernel Compilation Flow

Compiling OpenCL kernels to generate hardware configuration can take up to multiple hours. Thus, Altera OpenCL Compiler (AOC) provides useful approximations and emulation techniques which approximate the behaviour of a kernel on a targeted FPGA board in just a few minutes. The details of the kernel compilation flow are shown in figure 2.6 [29]. One important step in the flow below, is checking if the kernel is **single-work-item**. This is an important concept on FPGA optimization and is fully discussed in the next chapter.

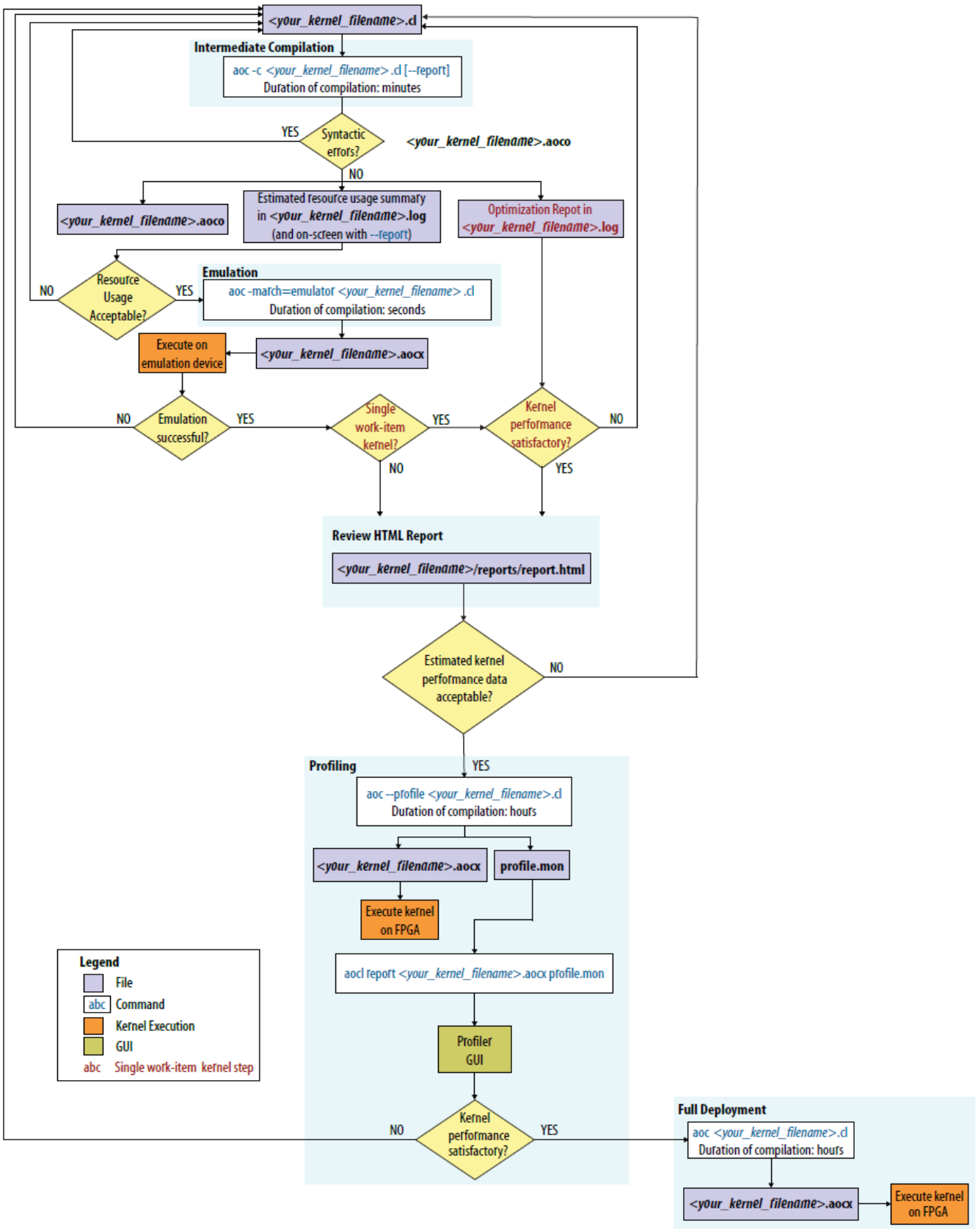


Figure 2.6: Intel FPGA SDK kernel compilation flow

An estimation of the resources, throughput due to control flow analysis, global memory bandwidth, resource usage summary and optimization suggestions is given by $\boldsymbol{aoc\ -g\ -c} < \boldsymbol{kernel\_filename} >.\boldsymbol{cl}$ command while checking for syntax errors of the kernel code. **-c** argument generates an Altera Offline Compiler Object file ($.aoco$) file without generating the hardware build. **-g** command generates the optimization report with corresponding line numbers in the kernel. The kernel can then be executed by emulation and its functionality can be analysed and estimated more accurately by $\boldsymbol{aoc\ -march = emulator} < \boldsymbol{kernel\_filename} >.\boldsymbol{cl}$ command. A full compilation can finally be done to generate the hardware configuration file called the Altera Offline Compiler Executable file ($.aocx$) file from the ($.aoco$) file by $\boldsymbol{aoc} < \boldsymbol{kernel\_filename} >.\boldsymbol{cl}$ command. The process takes hours to complete and Altera suggests doing this on a computer with at least 24 GB of RAM. The flow and the terms introduced above will constantly be used in the rest of the project.

2.2.3.2 FPGA pipelining

"Unlike GPUs, DSPs and multiprocessors, which achieve parallelism by replicating the same generic computation hardware multiple times, in FPGAs, however, we can achieve parallelism by duplicating only the logic the logic that algorithm exercises. This is done through a unique FPGA pipeline architecture" [30]. As a quick example, considering the OpenCL code:

```
size_t index = get_global_id(0);

C[index] = (A[index] >> 5) + B[index];
F[index] = (D[index] - E[index]) << 3;
G[index] = C[index] + F[index];
```

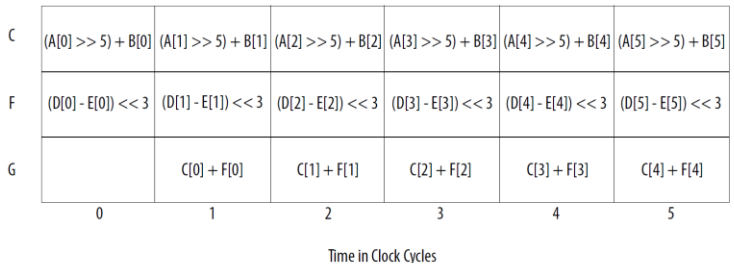


Figure 2.7: Pipeline FPGA execution with five instructions per clock cycle

Executing the above code takes several clock cycles on a CPU, GPU and DSP architecture models. However, an FPGA can execute the entire code in one clock cycle after the initial setup stage. Altera OpenCL Compiler will map and execute the code above to the pipeline model shown in Figure 2.7 [30]; The idea shown is that on each clock cycle different data enters the pipeline while every part of the pipeline is processing different (previous) threads. The parallelism comes from the fact that as data are streaming into the pipeline, different threads are being processed at the same time. In an FPGA OpenCL model, having each pipeline stage processing different data can be seen as executing different work-items in-flight.

2.3 KinectFusion

As mentioned before, SLAM algorithms build the map of the unknown environment and keep tracking the mobile-robot moving in that environment. The algorithms used in this project is called KinectFusion. As it appears from its name, this algorithm inputs the depth of an RGB-D camera like Microsoft Kinect, registers and fuses the stream of frames taken from different viewpoints into a 3D map [4]. A high-level overview of main steps and the kernels involved in the algorithm has been described below.

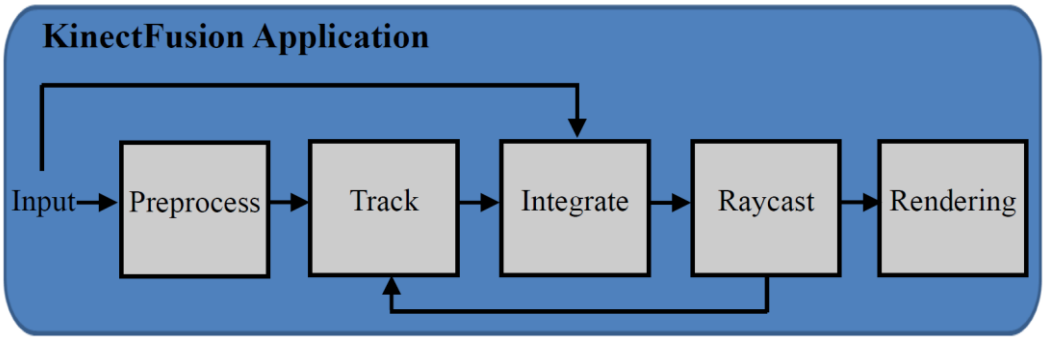


Figure 2.8 [4]: Key computational blocks of KinectFusion algorithm

- Pre-process: "KinectFusion normalizes each incoming depth frame and applies a bilateral filter before computing a point cloud (with normal) for each pixel in the camera frame of reference" [4].(*mm2meters* and *bilateralFilter* kernels).
- Tracking: estimates the current pose of the camera and registers it to the point cloud using iterative closest point (ICP) method. (*halfSample, depth2vertex vertex2normal, track, reduce* kernels)
- Integrating: After camera pose estimation, it's depth map is fused to the current 3D reconstruction (*integrate* kernel)
- Raycasting: recovering the 3D surfaces (*Raycast* kernel)
- Rendering: Visualisation step of KinectFusion algorithm (*renderDepth renderTrack* and *renderVolume* kernels).

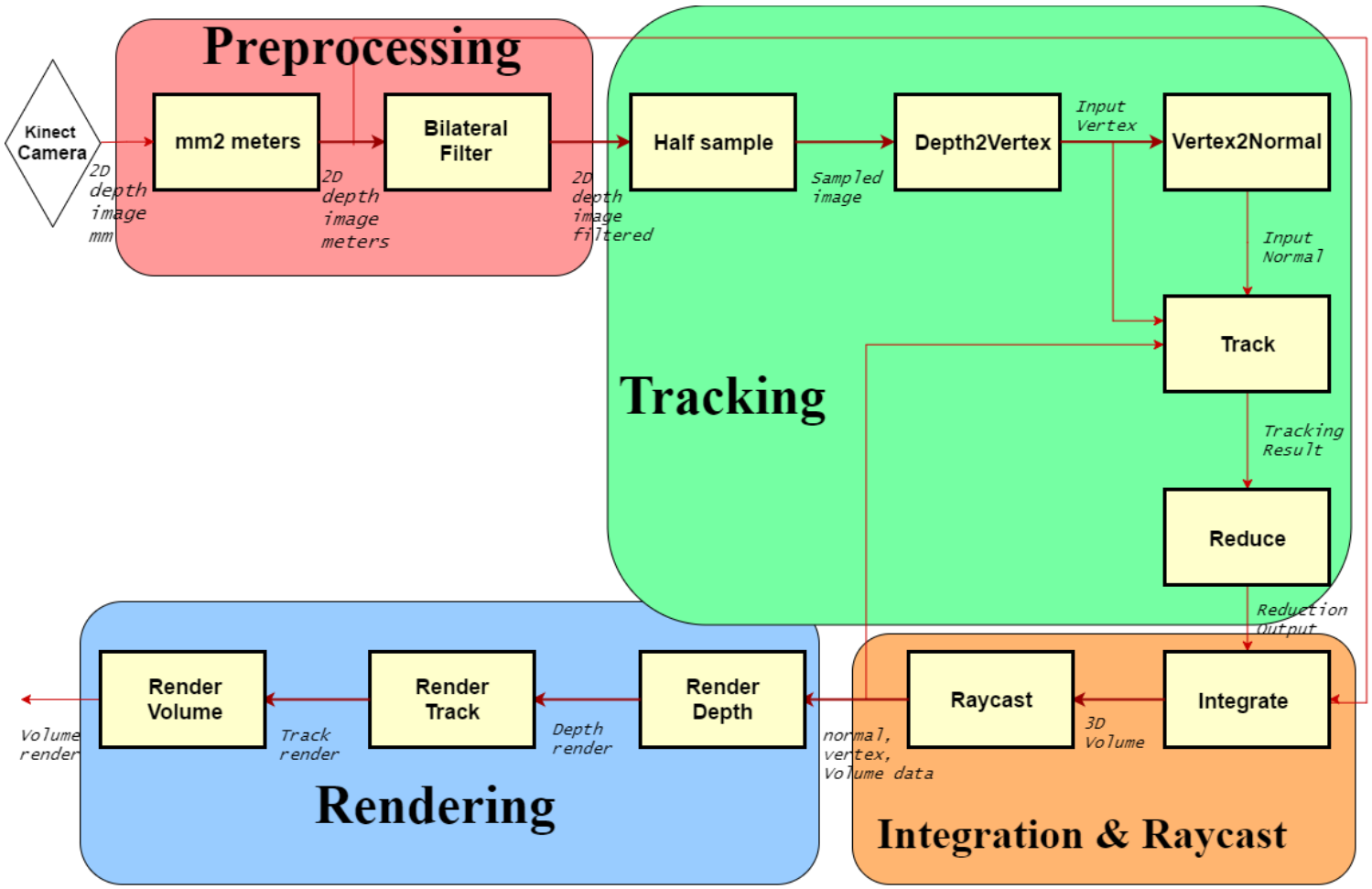


Figure 2.9: KinectFusion data flow with kernels and computational blocks

2.4 SLAMBench

Open source SLAMBench [4] implements KinectFusion in C++, OpenMP, OpenCL and CUDA. SLAMBench also provides techniques to evaluate the performance and energy-accuracy trade-offs of the KinectFusion algorithm targeted on desktops, laptops, mobile and embedded platforms. Input to SLAMBench can be standard formats(OpenNI), directly from the camera (RGB-D camera like Microsoft Kinect), raw format or ICL-NUIM dataset. This data set consists of 882 pre-recorded frames of a living room fed into the benchmark one after another, which is used in this project and is the accurate way of evaluating the SLAM implementation. “The structure of the code base allows for alternative kernels or algorithms to be plugged in with relative ease and, again, the effect on performance and accuracy to be easily analysed” [4]. What is important to highlight here is that SLAMBench has originally been written for CPUs and GPUs, and as mentioned before different hardware platforms like FPGAs require different tuning methods with OpenCL to maximise performance.

```
#!plain
-c  (--compute-size-ratio)       : default is 1   (same size)
-d  (--dump-volume) <filename>   : Output volume file
-f  (--fps)                      : default is 0
-i  (--input-file) <filename>    : Input camera file
-k  (--camera)                   : default is defined by input
-l  (--icp-threshold)        : default is 1e-05
-o  (--log-file) <filename>      : default is stdout
-m  (--mu)                       : default is 0.1
-p  (--init-pose)                : default is 0.5,0.5,0
-q (--no-gui)                    : disable any gui used by the executable
-r  (--integration-rate)         : default is 1
-s  (--volume-size)              : default is 2,2,2
-t  (--tracking-rate)            : default is 1
-v  (--volume-resolution)        : default is 256,256,256
-y  (--pyramid-levels)           : default is 10,5,4
-z  (--rendering-rate)   : default is 4
```

Input arguments and their default values in SLAMBench

SLAMBench can be used in 3 different modes:

1. Benchmark mode: This mode is used for benchmarking purposes and outputs elapsed time in each computational block, estimation of the camera position and a very useful indicator of whether camera tracking has been lost or not for each frame. In this project, this mode has used with living room trajectory 2 of ICL-NUIM dataset as input, outputting the mentioned parameters for the 882 frames in the dataset. This mode also allows each individual kernel timing as well.
2. Main mode: This mode is a GUI mode which uses GLUT for visualisation step and is not used for benchmarking purposes
3. Live mode: Used live with a depth sensor and again not suitable for benchmarking purposes.

It is worth mentioning that apart from the 12 kernels introduced previously, SLAMBench also uses 2 more kernels (*generateGaussian* and *InitVolume*) that are used just once in the initialization step.

3 Analysis and Design

Up to this point, the foundation of what is needed for this project has been introduced. That is in other words, OpenCL and how OpenCL works on FPGAs, Overview of KinectFusion algorithm and the SLAMBench framework. This section contains what approach is used, what metrics are used for evaluation, how they are measured and the optimization strategies applied for FPGA KinectFusion kernels.

3.1 approach

KinectFusion OpenCL has been used with living room trajectory 2 of ICL-NUIM dataset within SLAMBench framework throughout this project. Before moving on to the methodology, it is important to mention that KinectFusion computation in SLAMBench works in **process-every-frame** mode, where the next frame is only acquired when the previous frame computation is finished [4]. This ensures same scenes are executed on different platforms, regardless of their performance and whether real-time frame rates can be achieved.

The approach taken in this project can be described in the steps below:

1. Kernel preparation: This is the very first step needed to start the analysis. In this step, Firstly, SLAMBench has been tuned to compile for Intel OpenCL FPGA. Secondly, **per kernel** evaluation of application have been implemented. this is done by leaving the targeted kernel in the OpenCL version (.cl file), while replacing the rest of the application (kernels) with their C++ version. This allows the targeted kernel to run on the OpenCL platform, while the rest of the application runs on the host CPU. In this way, the performance of each part of the SLAM application (kernel) can be evaluated on different platforms, while enabling the optimization for FPGAs to be applied in per-single-kernel fashion. As part of this step, it was ensured the necessary data has been transferred between the host CPU and the kernel device to preserve overall SLAM functionality. This is further approved by SLAMBench output (Boolean camera tracking).
2. Starting from the first kernel, the performance of each kernel has been evaluated on CPU, GPU and FPGA device. The details of what devices have been used and how this is done has been stated in the next section, however the metrics of evaluation are:

a. **Execution time:** This is the time it takes to handle the targeted kernel. It involves the time it takes to write the data from host to the kernel device (write time), time for the kernel device to process the kernel (execution time) and finally the time it takes to get the data back to the host CPU (read time). The time reported is the average time of targeted-kernel calls of the 882 frames in the framework.
b. **Area:** This metric is only available to the FPGA device and as mentioned before, it is generated by the Intel FPGA SDK tool when the targeted kernel is compiled. The results shown here are not the estimation generated by the tool, but the actual area analysis when the kernel is fully synthesised on the FPGA (when the "aocx" file is generated) in the *.log* file.
c. **Absolute Trajectory Error (ATE):** This metric is provided by SLAMBench for accuracy evaluation of results. It performs the automatic testing against the ground truth from ICL-NUIM dataset. Values of 1-4.5 centimetres are acceptable for this parameter. SLAMBench [4] provides a Python program that calculates the mean total ATE once all 882 frames are processed.

3. Next, several FPGA optimization strategies have been applied to the OpenCL kernel, reporting how much the execution time has improved (speedup), while analysing how ATE and resources usage have been affected. These optimization strategies are described fully in the next section.
4. Strategies to grouping the kernels on the FPGA device have been analysed, proposing the best model, evaluating the whole design and where it stands in the SLAMBench framework when compared with other SLAMBench platforms in terms of accuracy (ATE), frame rate and power efficiency.

3.2 Platforms

Details of the platforms named in the analysis are as follows:

1. **FPGA:** The FPGA board targeted is *Nallatech 385* with *Stratix V D5 FPGA* which is hosted on *ee-snowball2* server. This FPGA has two banks of SDRAM, each have 4 GB, x72, DDR3 SDRAM running at 1600 MT/s. This FPGA uses 8 lane PCI Express.
2. **CPU OpenCL:** The CPU OpenCL platform is hosted on *ee-snowball2 server*. This machine has **four** Intel Core i3-2130 CPUs running at 3.40GHz.
3. **CPU C++:** This is the pure C++ version of algorithm (kernels) running sequentially in *ee-snowball2* server on Intel Core i3-2130 CPU.
4. **CPU laptop:** This the C++ version of algorithm running sequentially on *surface pro 3* machine, which has four Intel Core i5-4300U CPUs running at 1.90GHz.

5. **GPU:** This GPU is of type AMD FirePro™ W5000 Graphics card hosted on *ee-snowball0* server.

The compilation of OpenCL on FPGAs has been done in *ee-boxer4* server (because of high number of cores), which shares the same directory as the *ee-snowball2* server. These servers belong to Imperial College London. Altera OpenCL SDK installed on the servers is version 15.0.0.145 build 145. This version was updated towards the end of the project to version 16.1.0.196 Build 196, in which the name is changed to Intel(R) FPGA SDK for OpenCL.

3.3 FPGA optimization

When structuring OpenCL kernels for FPGA, it is fundamental to understand two models:

1. **NDRange kernels:** This is the typical OpenCL kernel described previously where there exists a global work size, work groups and work-items partitioned over the index space. "This model is defined as *Data Parallel Programming Model* and there is a mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel" [21]. As mentioned before, this model best matches to the architecture of GPUs and CPUs. Original KinectFusion OpenCL implementation of SLAMBench kernels are all written in NDRange format.
2. **Single-work-item kernels:** This model is known as *task parallel programming model* in OpenCL specification, where a single instance of the kernel is executed independent of the index space [21]. It can be seen as executing NDrange kernels but with index size of (1,1,1). When designing kernels in single-work-item, the Intel FPGA SDK for OpenCL exploits loop pipelining concept introduced earlier in the report. Thus, this model best matches to the architecture of FPGAs. "Kernels that do not use any work-item built-in functions, such as *get_global_id()*, are compiled for single-work-item execution (a task). Otherwise, a kernel is compiled as an ND-Range. For NDRange kernels, the loops are not pipelined. Instead, they are built to accept multiple work-items simultaneously. NDRange kernels can achieve reasonable performance on FPGAs but usually A large number of threads is required to do so" [30].

Just to provide an example, here is how our first kernel (mm2meters) can be changed to single-work-item. format.

```
__kernel void AOCmm2metersKernel(
    __global float * depth,
    const uint depthSize_x ,
    const uint depthSize_y ,
    const __global ushort * in ,
    const uint inSize_x ,
    const uint inSize_y ,
    const int ratio )
{
  uint2 pixel = (uint2) (get_global_id(0),get_global_id(1));
  depth[pixel.x + depthSize_x * pixel.y] = in[pixel.x * ratio + inSize_x * pixel.y * ratio] / 1000.0f;
}
```

Figure 3.1: mm2meters kernel in NDRange model

```
//To instruct AOCL to compile in single-work-item fashion
__attribute__((task))
__kernel void AOCmm2metersKernel(
    __global float * restrict depth,
    const uint depthSize_x ,
    const uint depthSize_y,
    const __global ushort * restrict in ,
    const uint inSize_x ,
    const uint inSize_y
    const int ratio )
{

  //functionality is the same as -> pixel = (uint2) (get_global_id(0),get_global_id(1)
  for(uint pixel_y=0;pixel_y<depthSize_y;pixel_y++){
    for(uint pixel_x=0;pixel_x<depthSize_x;pixel_x++){
      depth[pixel_x + depthSize_x * pixel_y] = in[pixel_x * ratio + inSize_x * pixel_y * ratio] / 1000.0f;
    }
  }
}
```

Figure 3.2: mm2meters kernel in single-work-item model

As can be seen the *get_global_id()* function must be replaced with for-loops and *__attribute__(task)* is needed to be declared. The host side kernel execution command should also be changed from *clEnqueueNDRangeKernel()* to *clEnqueueTask()* function.

At this point, it is worth mentioning that a similar MSc project to this project was done by Sergio Iannace in 2015 [31] in Imperial College London. His approach was executing all kernels in NDrange fashion, which resulted in a poor performance on the FPGA. The unique approach to this project is loop-pipelining the kernels, as suggested by Altera. In fact, Altera suggests "designing your OpenCL application as a single work-item kernel is sufficient to maximize performance", explicitly focusing on importance of loop-pipeline parallelism approach.

The next section describes the optimization techniques introduced in Altera OpenCL best practice guide and Altera OpenCL optimization guide [30] [29] that has been applied to OpenCL KinectFusion kernels.

3.3.1 Optimizing single-work-item kernels

As described in the compilation flow, by inferring ($aoc\ -g\ -c\ <kernel\_name>.cl\ -o\ kernel\_name.aoco$) command, an optimization report is generated (HTML report is generated in the newest version of SDK instead). Intel FPGA SDK for OpenCL generates this optimization report **only** for single-work-item kernels. The report illustrates the status of all the loops within the kernel (is pipelined or not) and any performance bottlenecks that stop the kernel to be pipelined with some suggestions on how to resolve them.

```
================================================================================================================
Kernel: AOCmm2metersKernel
================================================================================================================
The kernel is compiled for single work-item execution.

The kernel has a required work-group size of (1, 1, 1).

Loop Report:

 + Loop "Block1" (file mm2_meters.cl line 49)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Loop "Block2" (file mm2_meters.cl line 50)
     Pipelined well. Successive iterations are launched every cycle.


================================================================================================================
```

Figure 3.3: optimization report generated for single-work-item mm2meters kernel

What is important to mention here is the terminology of Initiation Interval (II). This means how many number of clock cycles the FPGA stalls until it can process the next iteration of the loop. For the simple case above, the most ideal II of one (launched every clock cycle) is achieved. Therefore, the loop is pipelined well. In many cases, the tool reports variable, memory, loop carried dependencies or many other factors that stall the pipeline, resulting in a higher II than one. In fact, the throughput is evaluated using the formula:

$$Throughput = \frac{Kernel\ frequency\ x\ block\ size}{pipleline\ stalls}$$

Where kernel frequency is fixed (220 MHz in our case) and block size is the size of data being analysed. What can be seen that to achieve the maximum ideal performance (highest throughput), there should be no stalls in the pipeline.

Below techniques has been used to improve the performance of single-work-item kernels in this project, after converting them to this model, in order to maximise pipelining. Details of the implementation are shown when going through each individual kernel in the next chapter.

- **Using one single loop instead of many:** combining nested loops in to a single loop reduces the hardware footprint and overhead between the iterations. [30].
- **Direct Memory Access (DMA) transfer:** Host side buffers should be at least 64-byte aligned. In this way DMA transfer can be occurred to improve host-kernel transfer time.
- **Host side calculation:** In some cases, (a good example later in *bilateral filter* kernel), some constant calculations can be performed in the host side CPU and gets passed as a kernel argument to the kernel. This is particularly useful for operations that are expensive on an FPGA architectures like integer divisions and most floating point operators. If host side calculation cannot be done, cheap FPGA operators that consume low hardware resources like binary logic operations and shifting by constants is used if applicable [30].
- **Avoiding loop branching:** NDRange kernels usually contain many branches and *return* statements within the loops, that stall the pipeline. Restructuring the kernel to avoid these loop branching can ensure a well pipelined flow.
- **Reducing global memory access:** Off chip memory access is a crucial and expensive operation in OpenCL design on FPGAs. Pre-loading data to local and private memory to avoid SDRAM several accesses if possible, can help improve the overall performance.
- **Unrolling loops:** This method can be used by declaring $\#pragma\ unroll$ above loops to touch the performance-area trade-off on FPGAs. The FPGA does less number of iterations at the expense of increased hardware resources [30]. If full unrolling is not possible, partial unrolling has been implemented by introducing an unroll factor.

- **Avoiding pointer aliasing:** By including the *restrict* keyword for pointer arguments, Intel FPGA SDK for OpenCL compiler prevents unnecessary memory dependencies among non-conflicting load and stores [30].
- **Avoiding out of order loop iterations:** This is done by designing inner loops that their upper and lower bound are constant or does not depend on the outer loop iteration space [30].
- **Declaring variables in the deepest scope:** This is done to reduce the hardware resources by making the compiler not preserve variable data in loops that do not get used**.**
- **Loop-carried dependencies:** When the feedback from the optimization report suggests that there are data/memory dependencies in the loops that stall the pipeline, a set of techniques can be applied to remove/relax these dependencies. These techniques include inferring shift registers, transferring the loop-carried dependency to local memory and implementing simpler memory access patterns [30], which has been applied to some kernels in our application. In this way, the report states that the dependency is removed and full pipeline execution is inferred.
- **Asynchronous command queues:** command queues (write, execution and read commands in OpenCL) can be blocking or non-blocking. The default KinectFusion OpenCL implementation uses blocking command queues, which means that the host CPU control continues the serial execution only if the OpenCL command is finished (i.e. memory transfer is completed). A better approach is passing dependency flags between OpenCL commands, and changing them to non-blocking commands. In this case, the host CPU can continue its execution, and check for the dependency flags when it needs to execute the next OpenCL command queue, allowing operations to happen concurrently on the host, while the host-kernel transfer is being made.

3.3.2 Optimizing NDRange kernels

Some dependencies are not possible to resolve, or there may exist barriers in the kernel that stall the pipeline by a large amount and cannot be removed. For these kernels, it is better to execute them in NDRange model on the FPGA device. As mentioned before, for this model, large number of threads are required to achieve a good performance. The scope of their optimization is not as much as single-work-item kernels, nevertheless most of the optimizations described above can be applied to them as well. there are some techniques that can be applied to NDRange kernel only to improve their data proficiency as introduced in Intel FPGA SKD Best Practice Guide [30].

- **Specifying work-group sizes:** This is done by $reqd\_work\_group\_size(X,Y,Z)$ attribute, which makes the compiler perform aggressive optimizations to match the kernel to hardware [30].
- **Kernel vectorization:** This method allows multiple work-items to execute in a single instruction multiple data (SIMD) manner. It is specified by $num\_simd\_work\_items(N)$ where N is the vectorization factor. By implementing N SIMD vector lanes, every work-items performs N times more work [30].
- **Multiple compute units:** by specifying $num\_compute\_units(N)$ attribute, Intel FPGA SDK offline compiler can generate N compute units, where each can execute multiple work groups simultaneously. This results in a higher throughput [30].
- **Combining kernel vectorization and multiple compute units:** In cases where the vectorised or replicated design does not fit the FPGA, having both vectorization and compute unit factor helps to reduce the area, while increasing the overall throughput [30].

3.4 Methodology

The optimization approach taken for per-kernel FPGA optimizations is loop-pipelining the kernels by converting them to single-work-item and apply the single-work-item optimization strategies described above. If after applying all the single-work-item optimizations, the report states that there still exists performance bottlenecks i.e. the loops are not fully pipelined, the NDRange optimization approach is taken instead.

The default values of SLAMBench has been chosen for optimization and kernels timing to allow a fair final evaluation in the benchmark. That is "the integration rate is set at 2 frames, the voxel volume is 256x256x256, the volume size is 4:8x4:8x4:8 m3, rendering rate is set at 4 frames and the input image is halved during the preprocessing, i.e. 320x240. All the experiments are run on the living room trajectory 2 of the ICL-NUIM dataset" [4].
Last argument of command-queue functions (*clEnqueueWriteBuffer, clEnqueueNDRangeKernel, clEnqueueTask, clEnqueueReadBuffer*) are event flags associated with the corresponding enqueue commands. Kernel timing is measured by passing that event to the *computeEventDuration(cl_event event)* function provided by Intel FPGA library. It is important to highlight that using a standard timer and "ticking" them after write, kernel execution and read functions above does not generate accurate timings. With regards to C++ timings platforms however, the timings are taken by the *chrono* timer. As mentioned before, the timings are the average kernel timings of all the frames reported all in milliseconds. In case a kernel gets called more than once per frame, again the average of all the calls have been measured. For the FPGA timings, the results shown are those when the kernels are fully

synthesized (after hours' compilation) and run on the FPGA on the server. The emulator (shown in the compilation flow) executes the kernel in serial and is in any of the measurements. The ATE is measured by running a Python program once all 882 frames are finished. The value reported here is the total mean ATE. The area is also generated by the tool in the *.log* file in the directory of compilation after hardware generation is completed.

All the detailed code implementations with comments can be found on the GitHub repository in [32]. The notes and comments there fully explain what each folder contains along with anything which has been implemented. For the purpose of this report, the most important parts of implemented codes are shown.

4 implementations

4.1 SLAMBench tuning

Tuning SLAMBench to compile for FPGAs was one of the challenges faced after per-kernel code was written. As mentioned previously, SLAMBench is written for CPUs and GPUs OpenCL and although the language is OpenCL, FPGAs require many additional tasks to run OpenCL kernel code. This includes not only reading from the *.aocx* file, but several libraries, headers and FPGA run-times flags that need to be included as part of the build. This makes the problem extremely difficult in a complex build like SLAMBench, where several CMake files call each other recursively, to build the whole application for many programming languages in one build. The initial approach to this, was doing the whole FPGA design completely out of SLAMBench directory.

This was done by writing each kernel's host-kernel code (by replacing "hello-world" example provided by Altera), where compiling and running on FPGAs did not have any issues. In this approach, the input pixels of a given frames were written to a text file, the host was programmed to read the input pixels from a file, process the targeted kernel on the FPGA and write the output pixels' data back to a text file, integrating FPGA in SLAMBench flow. To check the accuracy, the output pixels of GPU in SLAMBench was compared to the output text files here. The FPGA optimization was also done afterwards, again comparing the suggested parameters. For all 12 kernels this task was done. The code for this can be found in the Github repository [32]. The architecture of this model can be shown below:

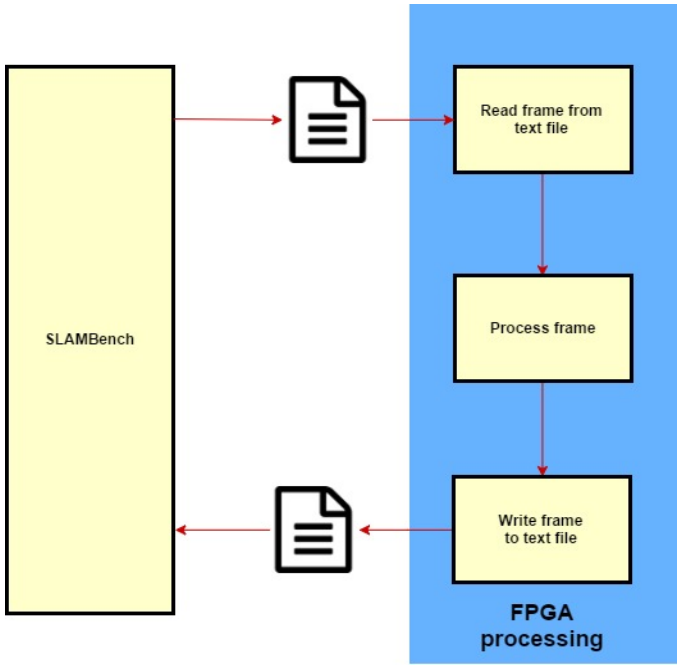


Table 4.0.1: Designed model for SLAMBench integrated with FPGA outside the framework

It is clear that this approach is not ideal due to the following reasons:

- Synchronizing SLAMBench and FPGA processing is challenging, since frames are not processed in the same amount of time on the FPGA.
- ATE cannot be measured, since all processing is done outside SLAMBench directory.
- Extremely time consuming; since the whole SLAMBench framework is essentially being written for FPGAs.

After taking the timing measurements here (for one frame only) for all kernels, in the second attempt, SLAMBench was successfully tuned to be compiled for FPGAs with assistance from the circuit research group. This was done by injecting all Intel FPGA SDK libraries in a newly written CMake file, and changing all initialization function to FPGA OpenCL style format (to read from aocx file and call the *createProgramFromBinary* ()) function instead. Hence, the FPGA results shown below are (as described in the approach section) within the ideal SLAMBench framework for FPGA (e.g. taking the average execution time of all frames) just like other platforms.

4.2 kernels

For each kernel, a brief description of what the kernel does, hardware diagram of the data flow and timings in milliseconds to 3 decimal places has been reported. “write time” is the host-to-kernel transfer time, “execution time” is the kernel execution time on the hardware accelerator, and “read time” is the kernel-to-host read back time. “Total time” for each kernel is sum of all the timings (write time + execution time + read time). All optimizations applied to individual kernels have been described in detail. “FPGA optimized” timing is the timing once all the stated optimizations have been applied to the kernel. The ATE of all kernels have also been reported in meters. With regards to hardware diagram, this is generated by the compiler in the newer version of SDK and can be found in the HTML reports once the original kernel is compiled for FPGA. Only overall important sections of the diagram (marked in red) is explained. The full HTML files with details of every block can be found in the git repository [32].

4.2.1 mm2meters

This kernel changes the 2D depth image from millimetres to meters.

```
__kernel void AOCmm2metersKernel(
    __global float * depth,
    const uint depthSize_x ,
    const uint depthSize_y ,
    const __global ushort * in ,
    const uint inSize_x ,
    const uint inSize_y ,
    const int ratio )
{
  uint2 pixel = (uint2) (get_global_id(0),get_global_id(1));
  depth[pixel.x + depthSize_x * pixel.y] = in[pixel.x * ratio + inSize_x * pixel.y * ratio] / 1000.0f;
}
```

Figure 4.0.2: original mm2meters kernel

Since this kernel is a simple kernel, only a few optimizations techniques could be applied. These are:

- Loop pipelining: converting to single-work-item model.
- Avoiding pointer aliasing: on variables *depth* and *in.*
- Asynchronous command queues.
- The hardware diagram as shown in Figure 4.0.4 contains only one block, where each work-item (pixel) is accessed from global memory, processed and stored back to global memory again.
- Direct memory access (DMA) transfer: With regards to transfer time between the host and kernel, this alignment optimization is the only optimization that can be done, which has a significant impact as shown in figure 4.0.3.

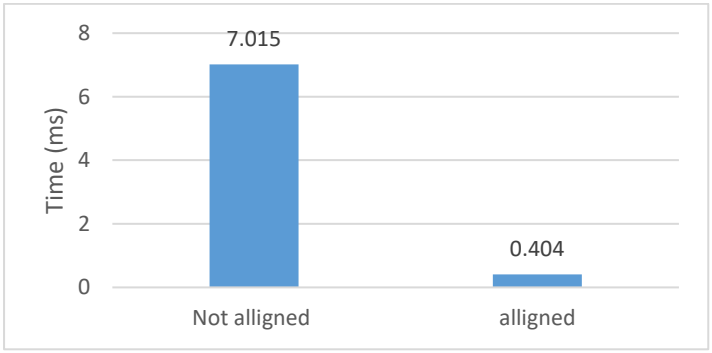


Figure 4.0.3: mm2 host-kernel transfer time (ms)

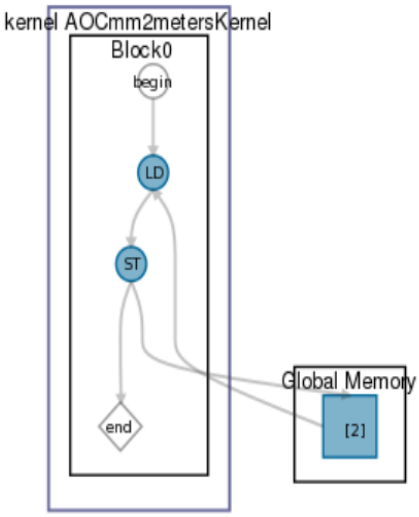


Figure 4.0.4: mm2meters kernel hardware diagram

```
__attribute__((task))
__kernel void AOCmm2metersKernel(
    __global float * restrict depth,
    const uint depthSize_x ,
    const uint depthSize_y ,
    const __global ushort * restrict in ,
    const uint inSize_x ,
    const int ratio )
{
  for(uint pixel_y=0;pixel_y<depthSize_y;pixel_y++){
    #pragma unroll 5
    for(uint pixel_x=0;pixel_x<depthSize_x;pixel_x++){
      depth[pixel_x + depthSize_x * pixel_y] = in[pixel_x * ratio + inSize_x * pixel_y * ratio] / 1000.0f;
    }
  }
}
```

Figure 2.0.5: mm2meters optimized kernel

```
================================================================================================================
Kernel: AOCmm2metersKernel
================================================================================================================
The kernel is compiled for single work-item execution.

The kernel has a required work-group size of (1, 1, 1).

Loop Report:

 + Loop "Block1" (file mm2_meters.cl line 49)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Loop "Block2" (file mm2_meters.cl line 50)
     Pipelined well. Successive iterations are launched every cycle.


================================================================================================================
```

Figure 4.0.6: report for mm2meters optimized kernel

The report shows that full pipeline execution has been inferred for the optimized kernel for both loops as shown below.

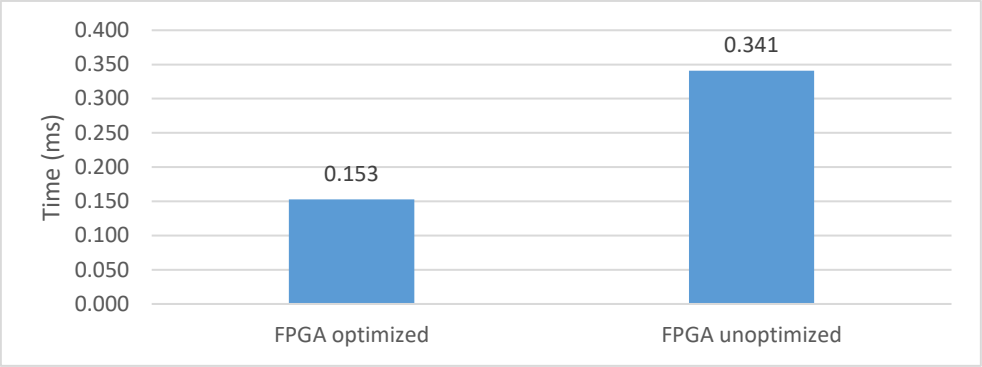


Figure 4.0.7: FPGA unoptimized vs optimized mm2 meters execution time (ms)

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.210			0.404	7.015
Execution time	0.145	0.054	0.465	0.370	0.153	0.341
read time		0.149			0.317	0.317

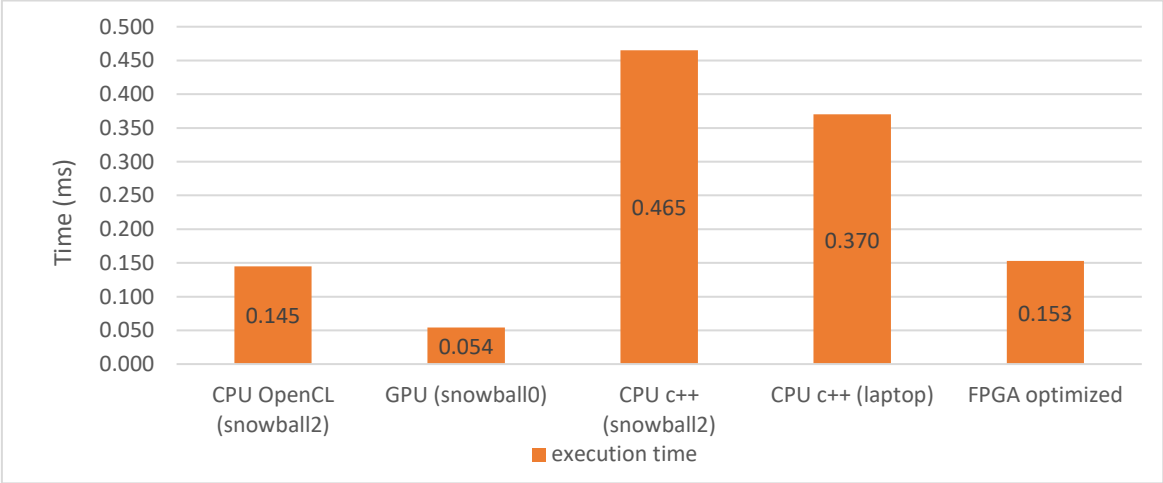


Figure 4.0.8: mm2meters timings (ms) for FPGA optimized and all other platforms

Analysis:

- Write time and read time is only sensible for non-CPU OpenCL platforms, which are FPGA and GPU. What is worth mentioning here, is that FPGA optimized has twice as GPU transfer time. The reason for this is that the targeted FPGA board uses 8 lane PCI express (OpenCL Gen2). The GPU however, uses 16 lane PCI express, transferring data twice as fast as from and to the host CPU.
- It is important to mention here that, it only makes sense to send data back and forth to the OpenCL device, only if the total time of doing this so is faster than sequentially

doing it on the host CPU. This is what is analysed later in the report and this section only analyses the per-kernel KinectFusion timings on different platforms.

- With optimizations applied, the total FPGA time (write, execution and read) is 8.7 times faster than the unoptimized FPGA version. The execution itself has seen 2.2 times speed up.
- The pattern that can be seen is that the optimized FPGA is much faster (up to 3x) than the sequential C++ CPU on laptop and snowball2 machines.
- As expected, the CPU OpenCL is 4x faster than the C++ version on the same machine, taking advantage of 4 cores available.
- The focus of timings from now is going to be comparing the optimized FPGA with multi-core CPU (CPU OpenCL) and GPU. The execution time of FPGA is almost the same as CPU OpenCL and slightly slower than the GPU for this kernel. The scope of optimization here is not high and therefore, there is not much else that can be done.
- Due to the pragma unrolling factor introduced, there is an increase in the FPGA logic utilization and memory blocks resources usage.

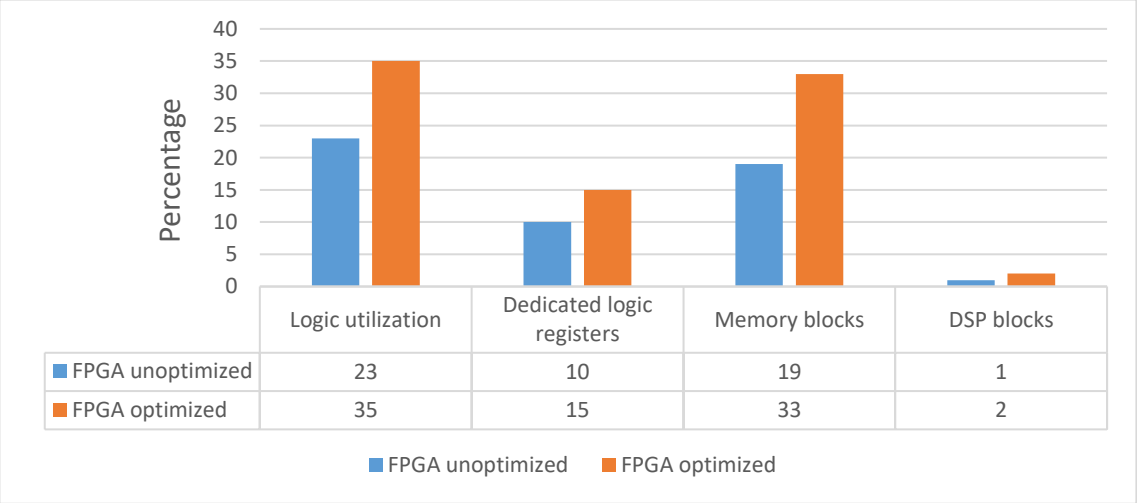


Figure 4.0.9: % of FPGA area usage mm2meters optimized vs unoptimized

The ATE of the results (meters) are shown below:

Platform	FPGA unoptimized	FPGA optimized
ATE (meters)	0.0206	0.0206

The ATE of both optimized and unoptimized versions are in acceptable range of 4.5 centimetres. The ATE has not changed, as the precision of none of the data types in the kernel has changed. ATE of the rest of the kernels are shown and discussed in the results section.

4.2.2 Bilateral Filter

This is an edge-preserving blurring filter to reduce the noise and invalid input depths from the Kinect camera. The kernel essentially uses a window size of 5x5 and replaces the intensity of each pixel (based on distance) with the weighted average of intensity of nearby pixels, where weights are a Gaussian distribution.

```
86 inline float sq(float r) {
87   return r * r;
88 }
89
90 __kernel void AOCbilateralFilterkernel( __global float * out,
91     const __global float * in,
92     const __global float * gaussian,
93     const float e_d,
94     const int r ) {
95
96   const uint2 pos = (uint2) (get_global_id(0),get_global_id(1));
97   const uint2 size = (uint2) (get_global_size(0),get_global_size(1));
98
99   const float center = in[pos.x + size.x * pos.y];
100
101  if ( center == 0 ) {
102    out[pos.x + size.x * pos.y] = 0;
103    return;
104  }
105
106  float sum = 0.0f;
107  float t = 0.0f;
108
109  for(int i = -r; i <= r; ++i) {
110    for(int j = -r; j <= r; ++j) {
111      const uint2 curPos = (uint2)(clamp(pos.x + i, 0u, size.x-1), clamp(pos.y + j, 0u, size.y-1));
112      const float curPix = in[curPos.x + curPos.y * size.x];
113      if(curPix > 0) {
114        const float mod = sq(curPix - center);
115        const float factor = gaussian[i + r] * gaussian[j + r] * exp(-mod / (2 * e_d * e_d));
116        t += factor * curPix;
117        sum += factor;
118      }
119    }
120  }
121  out[pos.x + size.x * pos.y] = t / sum;
122 }
123
```

Figure 4.1.0: original bilateral filter kernel

Scope of optimization for this kernel is high. The optimization techniques applied to this kernel:

- Asynchronous command queues.
- Loop pipelining: After conversion to single-work-item, the report suggested that there are data dependencies on variable *t* and *sum*, which prevents fully pipelining the loops as shown in figure 4.1.1.

```
Iterations executed serially across the regions listed below.
Only a single loop iteration will execute inside the listed regions.
This will cause performance degradation unless the regions are pipelined well
(can process an iteration every cycle).

    Loop "Block4" (file bilateral_filter.cl line 125)
    due to:
    Data dependency on variable sum  (file bilateral_filter.cl line 121)

    Loop "Block4" (file bilateral_filter.cl line 125)
    due to:
    Data dependency on variable t  (file bilateral_filter.cl line 122)
```

Figure 4.1.1: Report indicating an II bottleneck due to the data dependencies

To solve this issue, the computation has been separated in two part. The first part, stores all the necessary additions in the array, and the second part adds them all up in an unrolled loop. In this way, in the expense of area, ideal II interval of one can be achieved with a high throughout.

- DMA transfer: 64-bit alignment of *in* and *out* host buffers
- Moving conditional statement: the conditional statement on top of the kernel *returns* if the pixel is black. This type of conditional statements (that return if condition is met), stalls the FPGA pipeline. To solve this, the *return* statement has been removed and the condition has been moved to the bottom of the kernel.
- Avoiding pointer aliasing: *in* and *out* variables.
- Declaring variables in the deepest scope possible.
- Host-side calculation: the result of *(2 * e_d * e_d)* is constant in the kernel, therefore the value is calculated on host and passed as an argument to the kernel.
- Reducing global memory access: The *gaussian* array is a constant small array, which gets accessed several times in global memory. Moving this array from global to private memory is safe and reduces the redundant slow access to global memory.
- Combining loops: The 2 nested for-loops of the window, have been combined to a single loop.
- Block3 in Figure 4.1.3 is the unoptimized diagram, highlighted because of the loop carried-dependency, resulting in latency of 400 ns in this block. The optimization reduced the latency to 64 ns.

```
================================================================================
Kernel: AOCbilateralFilterkernel
================================================================================
The kernel is compiled for single work-item execution.

The kernel has a required work-group size of (1, 1, 1).

Loop Report:

 + Loop "Block2" (file bilateral_filter.cl line 73)
 | Pipelined well. Successive iterations are launched every cycle.
 |
 |
 |-+ Loop "Block3" (file bilateral_filter.cl line 74)
   | Pipelined well. Successive iterations are launched every cycle.
   |
   |
   |-+ Fully unrolled loop (file bilateral_filter.cl line 85)
   |   Loop was fully unrolled due to "#pragma unroll" annotation.
   |
   |
   |-+ Fully unrolled loop (file bilateral_filter.cl line 103)
       Loop was fully unrolled due to "#pragma unroll" annotation.


================================================================================
```

Figure 4.1.2: generated report for optimized kernel

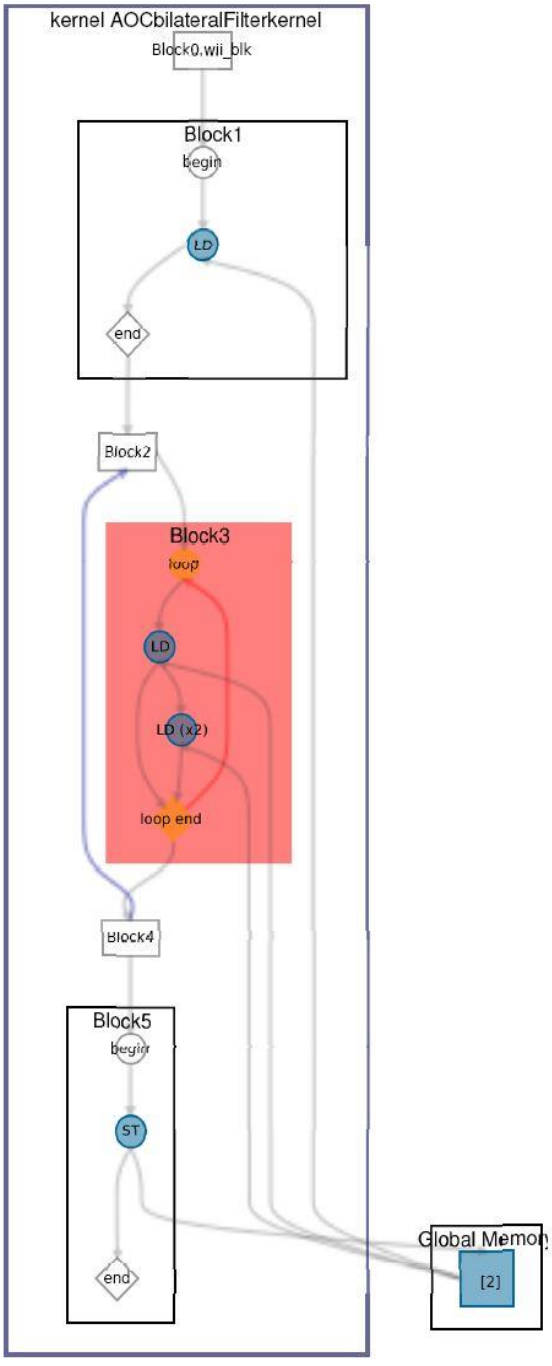


Figure 4.1.3: bilateral filter hardware diagram

The optimization report states that full pipeline execution has been inferred for all the loops in this kernel as shown in figure 4.1.2.

```
__attribute__((task))
__kernel void AOCbilateralFilterkernel( __global float * restrict out,
    const __global float * restrict in,
    const float mult_result,
    const int r ,
    const uint2 outSize) {
  const uint size_x=outSize.x;  //same as get_global_id(0)
  const uint size_y=outSize.y;  //same as get_global_id(1)

  float gaussian[6]={0.8824968934,0.9692332149,1.00,0.9692332149,0.8824968934};
  for(uint pos_y=0;pos_y<outSize.y;pos_y++){
        for(uint pos_x=0;pos_x<outSize.x;pos_x++){
            const float center = in[pos_x + size_x * pos_y];
            uint count=0;
            float t_array[25];    //accumulutate array of variable t
            float sum_array[25];  //accumulate array of varibale sum
            float sum = 0.0f;
            float t = 0.0f;

            int i=-3;   //i and j are chosen to match the previous values here and within the loop
            int j=-2;
            #pragma unroll
            for(int count = 0; count < 25; ++count) {
                //every 5 iterations increment i and set j back to -2. This ensures functionality is the same as having two nested loops
                if(count%5==0){
                  i++;
                  j=-2;
                }
                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 = (curPix - center)* (curPix - center);
                  const float factor = gaussian[i + r] * gaussian[j + r] * exp(-mod / mult_result);
                  t_array[count] = factor * curPix;
                  sum_array [count] = factor;
                }
                j++;
            }
           //removing loop dependency on sum and t
           #pragma unroll
           for(int j=0;j<25;j++){
             t += t_array[j];
             sum += sum_array [j];
           }
            out[pos_x + size_x * pos_y] = t / sum;
            if ( center == 0 ) {      //To remove the loop caried dependency, this if statement has been moved at the end of the kernel
            out[pos_x + size_x * pos_y] = 0;
            }
        }
    }
}
```

Figure 4.1.4: optimized bilateral filter kernel

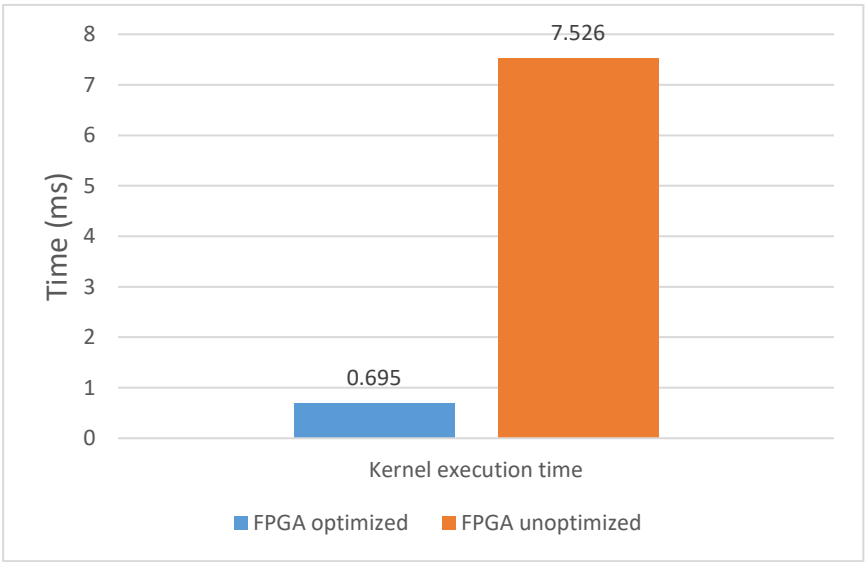


Figure 4.1.5: FPGA unoptimized vs optimized bilateral filter execution time (ms)

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.182			0.356	0.357
Execution time	5.213	1.002	44.462	75.283	0.695	7.526
read time		0.199			0.369	43.91

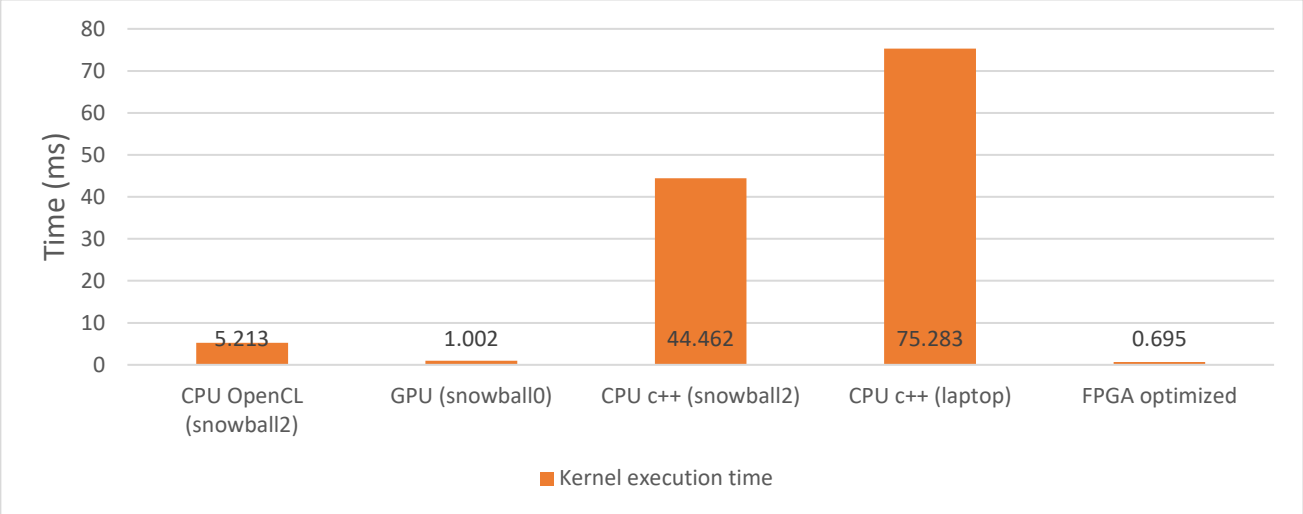


Figure 4.1.6: Bilateral filter timings (ms) for FPGA optimized and all other platforms

Analysis:

- The scope of optimization for this kernel was high and full pipeline execution has been inferred. The total time for optimized FPGA is 36 times faster than total time of the unoptimized version. The kernel execution time itself is 10 times faster than unoptimized execution time.
- Since this kernel matches best to the loop-pipeline parallelism architecture of FPGAs, the FPGA optimized execution time is almost 2x faster than the GPU and 7.5x faster than CPU OpenCL
- The strategy used to remove the dependencies and pragma unrolling factors resulted in a significant increase in the logic utilization and memory block usage of the targeted FPGA as shown in figure 4.1.7.

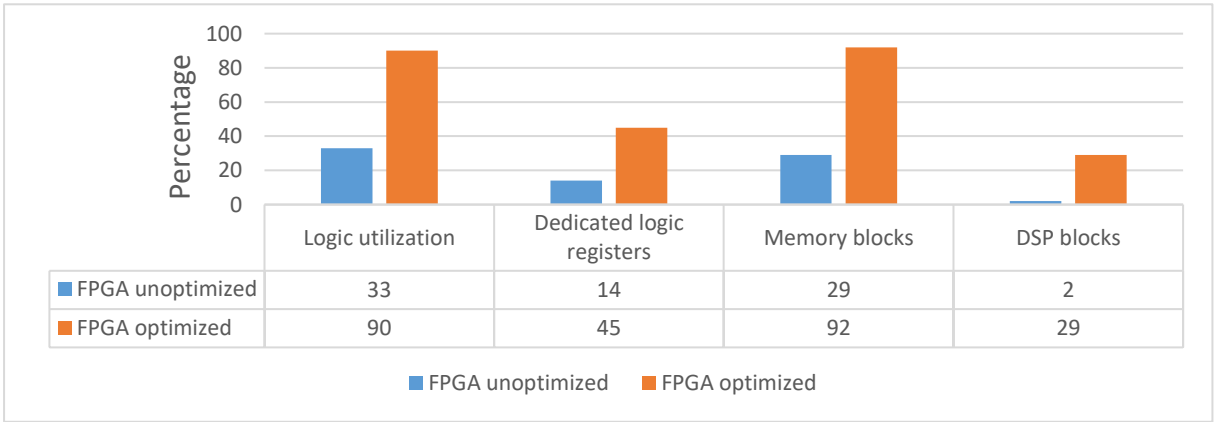


Figure 4.1.7: % of FPGA resource usage of bilateral filter optimized vs unoptimized

4.2.3 Half sample

This kernel samples the depth image and creates a three-level image pyramid [4]. The structure of this kernel is very much like the bilateral filter kernel. A window size of 4 is used to build the image pyramid.

```
134 __kernel void AOChalfSampleRobustImageKernel(__global float * out,
135     __global const float * in,
136     const uint2 inSize,
137     const float e_d,
138     const int r) {
139
140   uint2 pixel = (uint2) (get_global_id(0),get_global_id(1));
141   uint2 outSize = inSize / 2;
142
143   const uint2 centerPixel = 2 * pixel;
144
145   float sum = 0.0f;
146   float t = 0.0f;
147   const float center = in[centerPixel.x + centerPixel.y * inSize.x];
148   for(int i = -r + 1; i <= r; ++i) {
149     for(int j = -r + 1; j <= r; ++j) {
150       int2 from = (int2)(clamp((int2)(centerPixel.x + j, centerPixel.y + i), (int2)(0), (int2)(inSize.x - 1, inSize.y - 1)));
151       float current = in[from.x + from.y * inSize.x];
152       if(fabs(current - center) < e_d) {
153         sum += 1.0f;
154         t += current;
155       }
156     }
157   }
158   out[pixel.x + pixel.y * outSize.x] = t / sum;
159
160 }
```

Figure 4.1.8: original half sample kernel

Optimization techniques used in this kernel are as follows

- Asynchronous command queues.
- Loop-pipelining: After single-work-item conversion, the tool suggests that there is a dependency on variable *t* and *sum* stalling the pipeline by 8 clock cycles for each iteration of the loop. This dependency is highlighted red in Block3 in the hardware diagram (figure 4.1.9). **Shift-registers** have been inferred to remove the loop-carried dependency. In this approach, those values that need to be added up are shifted one after another to an array. Finally, all the elements of the array are added together (details commented in the code in figure 4.2.0).
- Avoiding pointer aliasing: adding *restrict* to *in* and *out* array.
- Declaring variables in the deepest scope possible.
- DMA transfer: on *in* and *out* host buffers.
- Combining nested loops to a single loop: Just like bilateral filter kernel, the nested loops (window) in line 148 and 149 (figure 4.1.8) have been combined to a single loop without additional price paid (since the modulo operator used is constant on one side).

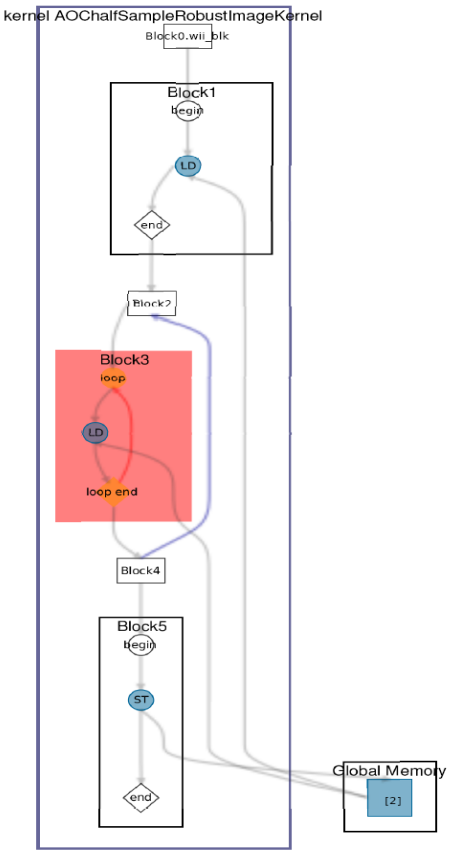


Figure 4.1.9: Half sample filter hardware diagram

- Loop unrolling: The single loop on line 159 (figure 4.1.8) can be fully unrolled.

```
__kernel void AOChalfSampleRobustImageKernel(__global float * restrict out,
    __global const float *restrict  in,
   const uint2 inSize,
   const float e_d,
   const uint2 outSize) {
for(uint pixel_y=0;pixel_y<outSize.y;pixel_y++){    //single-work-item
      for(uint pixel_x=0;pixel_x<outSize.x;pixel_x++){
          uint2 outSize = inSize / 2;
          const uint centerPixel_x = 2 * pixel_x;
          const uint centerPixel_y = 2 * pixel_y;
          float sum = 0.0f;
          float t = 0.0f;
          uint count=0;
          float shift_t[4];    //shift array t
          float shift_sum[4];  //shift array sum
          #pragma unroll
          for(int i=0;i<4;i++){  //zero initialization
            shift_sum[i]=0;
            shift_t[i]=0;
          }
          const float center = in[centerPixel_x + centerPixel_y * inSize.x];
           int i=-1;
           int j=0;
          #pragma unroll
          for(int count=0;count<4;count++){
              if(count%2==0){  //every 2 iteration i and j are are changed (combining loops)
                i++;
                j=0;
              }
              int2 from = (int2)(clamp((int2)(centerPixel_x + j, centerPixel_y + i), (int2)(0), (int2)(inSize.x - 1, inSize.y - 1)));
              float current = in[from.x + from.y * inSize.x];
              if(fabs(current - center) < e_d) {
                shift_sum[3] = shift_sum[0] + 1.0f;
                shift_t[3] = shift_t[0]+ current;
                #pragma unroll
                for(int j = 0; j < 4; ++j)      //shifting every element
                {
                  shift_sum[j] = shift_sum[j + 1];
                  shift_t[j] = shift_t[j + 1];
                }
              }
            j++;
          }

         #pragma unroll
         for(int j=0;j<4;j++){    //sum every element of shift register
           t += shift_t[j];
           sum += shift_sum[j];
         }
          out[pixel_x + pixel_y * outSize.x] = t / sum;
      `
```

Figure 4.2.0: optimized half sample kernel

The optimized kernel is fully pipelined with II of one.

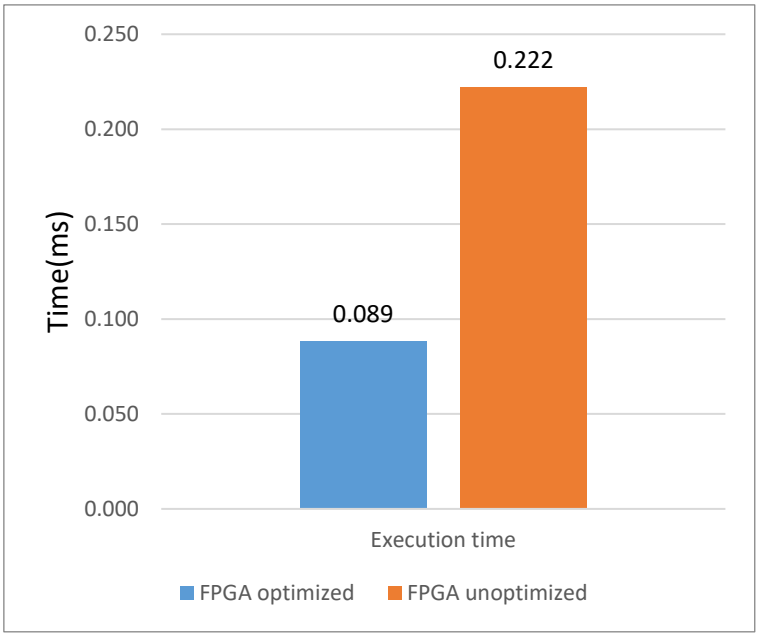


Figure 4.2.1: FPGA unoptimized vs optimized half sample execution time (ms)

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.102			0.230	3.491
Execution time	0.065	0.034	0.106	0.242	0.089	0.222
read time		0.080			0.141	0.136

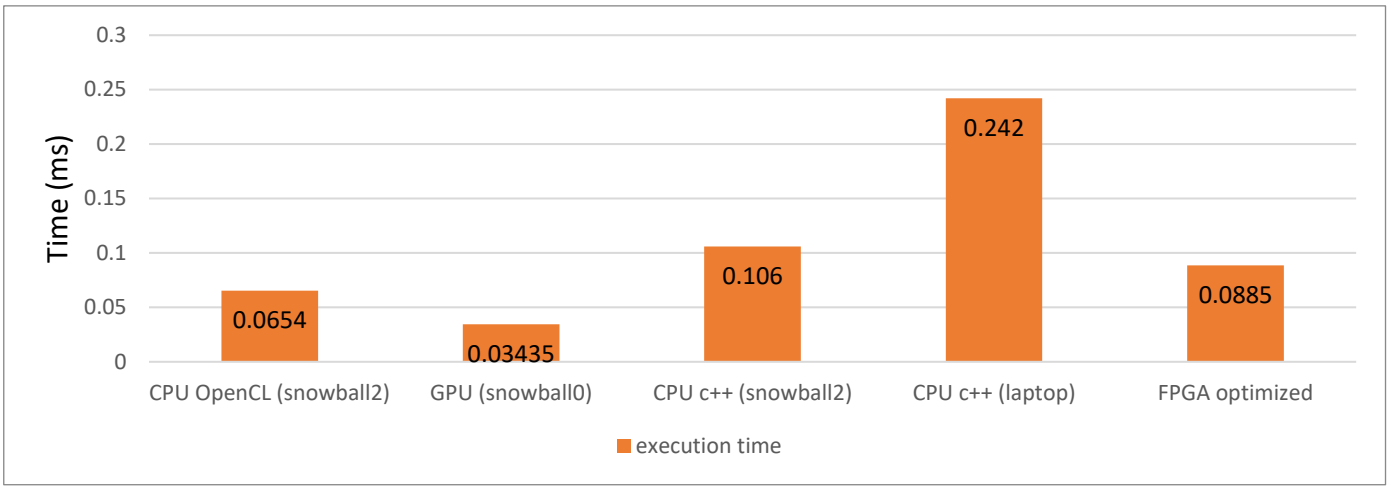


Figure 4.2.2: Half sample timings (ms) for FPGA optimized and all other platforms

Analysis:

- The optimizations applied resulted in 8.3 times speed up with regards to total time. The kernel execution time is 2.5 times faster than the unoptimized FPGA execution time.
- The shift-register approach has successfully removed the dependency and pipeline stall based on the optimization report.
- As before, due to the pragma unrolling factors introduced, there is a slight increase in the resource usage of this kernel on the FPGA device.

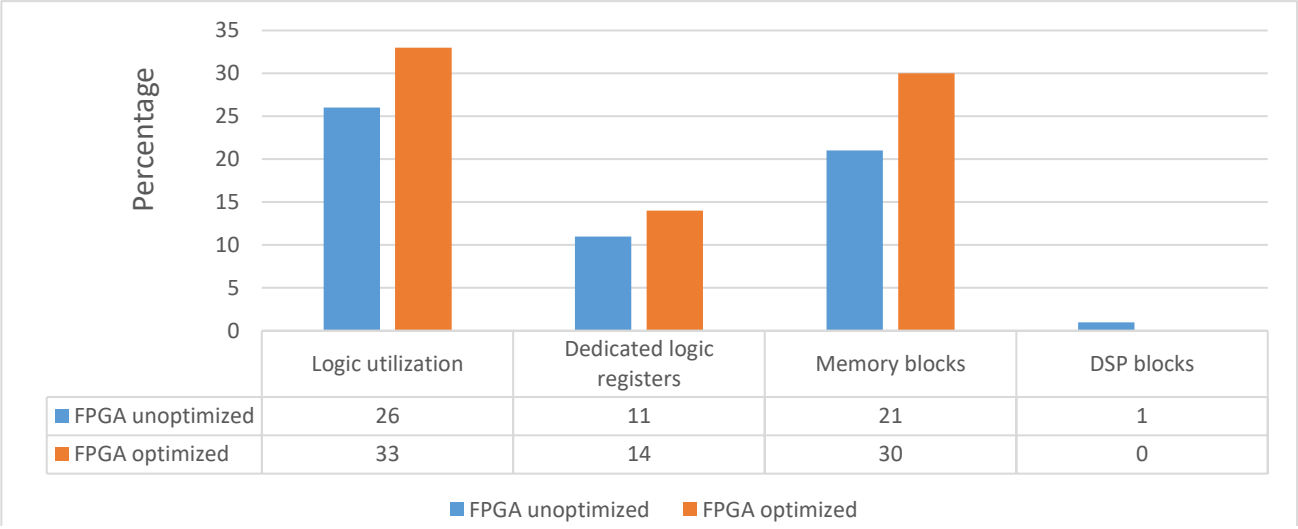


Figure 4.2.3: % of FPGA resource usage of half sample optimized vs unoptimized

4.2.4 Depth2vertex

This kernel changes each pixel of the depth image to a vertex (3D point) generating a point cloud. [4].

```
643 __kernel void depth2vertexKernel( __global float * vertex, // float3
644     const uint2 vertexSize ,
645     const __global float * depth,
646     const uint2 depthSize ,
647     const Matrix4 invK ) {
648
649   uint2 pixel = (uint2) (get_global_id(0),get_global_id(1));
650   float3 vert = (float3)(get_global_id(0),get_global_id(1),1.0f);
651
652   if(pixel.x >= depthSize.x || pixel.y >= depthSize.y ) {
653     return;
654   }
655
656   float3 res = (float3) (0);
657
658   if(depth[pixel.x + depthSize.x * pixel.y] > 0) {
659     res = depth[pixel.x + depthSize.x * pixel.y] * (myrotate(invK, (float3)(pixel.x, pixel.y, 1.f)));
660   }
661
662   vstore3(res, pixel.x + vertexSize.x * pixel.y,vertex);  // vertex[pixel] =
663
664 }
```

Figure 4.2.4: original depth2vertex kernel

Optimizations techniques applied to this kernels are:

- Asynchronous command queues.
- Avoiding pointer aliasing: on *vertex* and *depth* variables.
- Loop pipelining: After the conversion, the conditional if statement with *return* keyword stalls the pipeline (Since it is written in ND-range style). To fix this, the condition of the if statement has been reversed (De Morgan's law).
- DMA transfer
- Loop unrolling: the inner loop has been unrolled with a factor of 10.
- Struct arguments: One of the differences between Intel FPGA OpenCL and GPU-CPU OpenCL is the notation of structs. The typical structure data type written for CPU-GPU OpenCL will not work on an FPGA and is not supported by FPGA OpenCL compiler. To introduce structs on FPGAs using OpenCL, a pointer to global memory that points to that structure should be passed instead. For this kernel, this is done for *__global const Matrix4* restrict invK* argument, where *invK* is a pointer to *Matrix4* structure as shown in line 227 of the optimized code (figure 4.2.5).

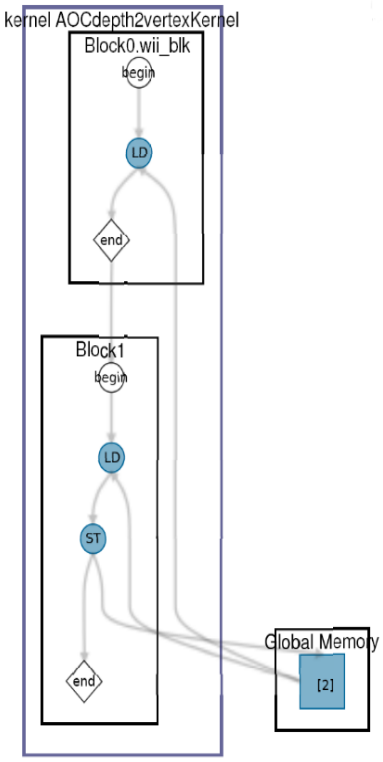


Figure 4.2.5: Depth2Vertex hardware diagram

- As shown in the hardware diagram (figure 4.2.5), there are 2 hardware blocks in this kernel. Block 0 is the condition check of if statement and block 1 is when the condition is met and the write back to global memory takes place. Since there are no loops, there is nothing stalling the pipeline.

```
typedef struct sMatrix4 {
  float4 data[4];
} Matrix4;


inline float3 myrotate(__global const Matrix4 * restrict M, const float3 v) {
  return (float3)(dot((float3)(M->data[0].x, M->data[0].y, M->data[0].z), v),
      dot((float3)(M->data[1].x, M->data[1].y, M->data[1].z), v),
      dot((float3)(M->data[2].x, M->data[2].y, M->data[2].z), v));
}



__kernel void AOCdepth2vertexKernel( __global float * restrict vertex, // float3
    const uint2 vertexSize ,
    const __global float * restrict depth,
    const uint2 depthSize ,
    __global const  Matrix4* restrict invK,
    const uint2 outSize ) {


  float3 vert= (float3)  (outSize.x,outSize.y,1.0f);

  for(int pixel_y=0;pixel_y<outSize.y;pixel_y++){
    #pragma unroll 10
    for(int pixel_x=0;pixel_x<outSize.x;pixel_x++){

        if(pixel_x < depthSize.x && pixel_y < depthSize.y ) {

        float3 res = (float3) (0);

        if(depth[pixel_x + depthSize.x * pixel_y] > 0) {
          res = depth[pixel_x + depthSize.x * pixel_y] * (myrotate(invK, (float3)(pixel_x, pixel_y, 1.f)));
        }
         vstore3(res, pixel_x + vertexSize.x * pixel_y,vertex);   // vertex[pixel] =
         //vertex[pixel_x + pixel_y * outSize.x]=res;
       }
    }
  }
```

Figure 4.2.6: optimized depth2vertex kernel

The approach to the kernel again, made it fully pipelined based on the report with II of one.

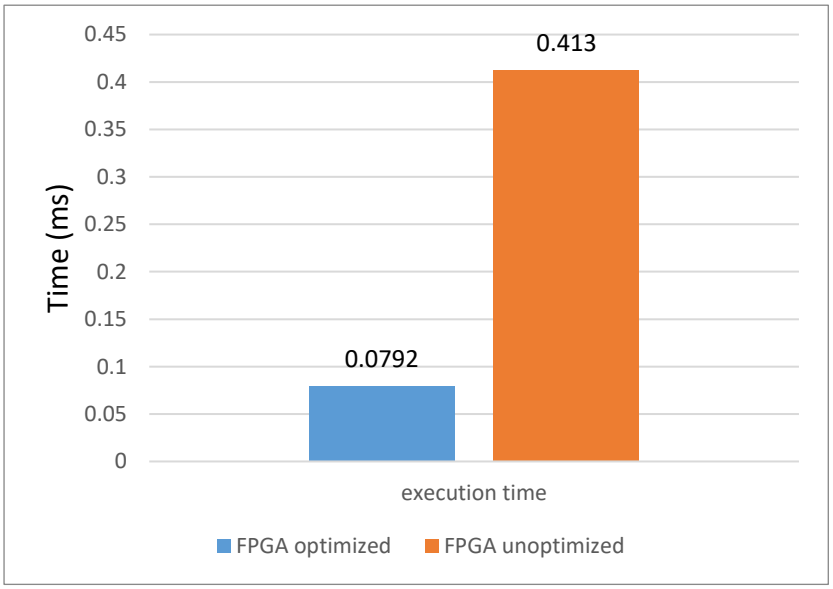


Figure 4.2.7: FPGA unoptimized vs optimized depth2vertex execution time (ms)

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.0628			0.141	0.142
execution time	0.105	0.0115	0.205	0.427	0.0792	0.413
read time		0.167			0.29	76.149

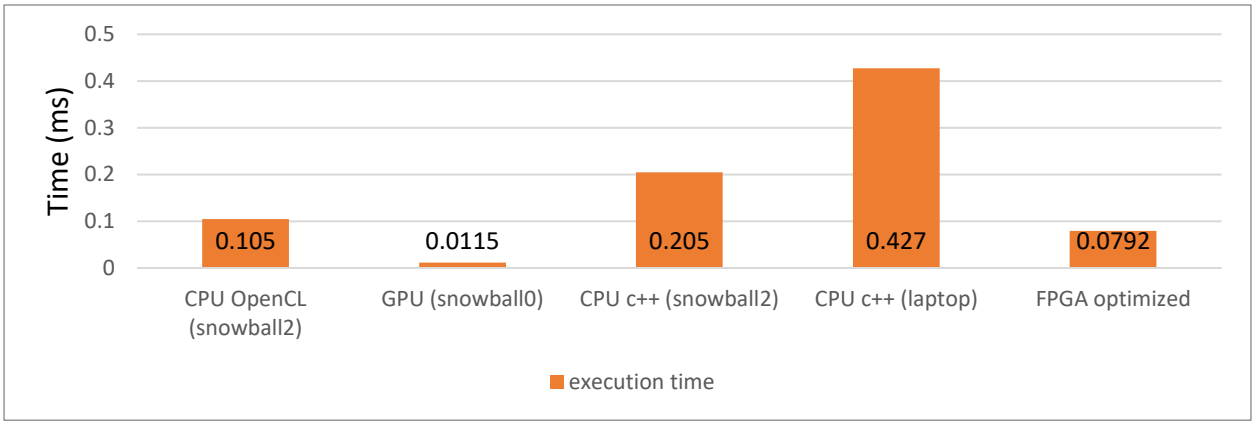


Figure 4.2.8: Depth2Vertex timings (ms) for FPGA optimized and all other platforms

Analysis:

- The total processing time of optimized FPGA is significantly faster than the unoptimized version due to the high read back time occurring in non-DMA transfer way. Execution time of the kernel is 5.2 times faster than the unoptimized version.
- FPGA optimized outperforms CPU OpenCL and sequential CPUs, however due to frequent slow off chip memory access of FPGAs, which is further increased by *structure* argument, GPU execution time is faster.
- Similarly, due to the ten unroll factor, FPGA optimized area has increased. Introducing the struct argument in global memory has also contributed in increasing the resources.

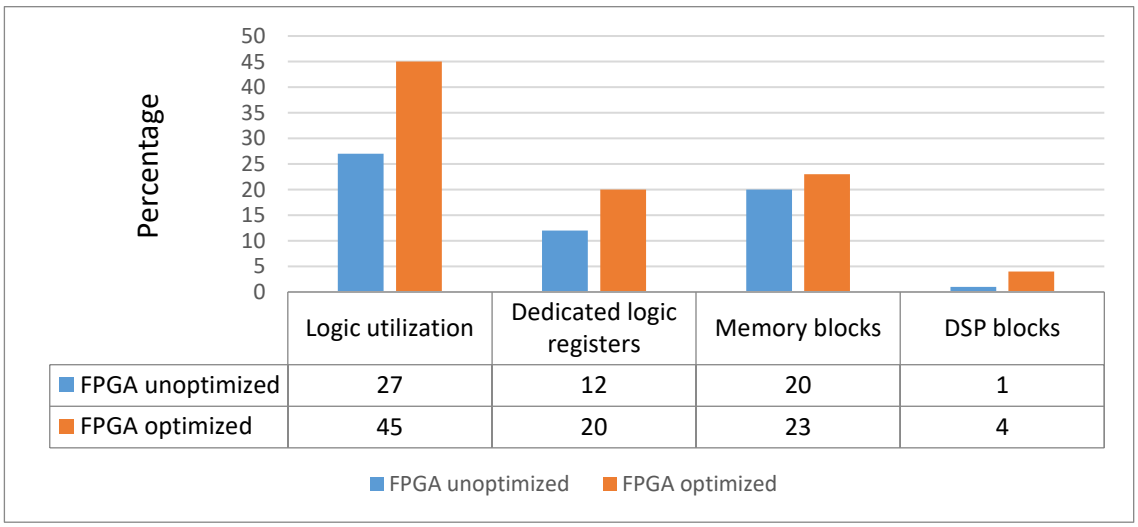


Figure 4.2.9: % of FPGA resource usage of depth2vertex optimized vs unoptimized

4.2.5 Vertex2Normal

This kernel computes the normal vectors for each vertex, which are used in ICP algorithm to calculate the point-plane distance between two vertices [4]. Original unoptimized kernel can be found in Appendix.

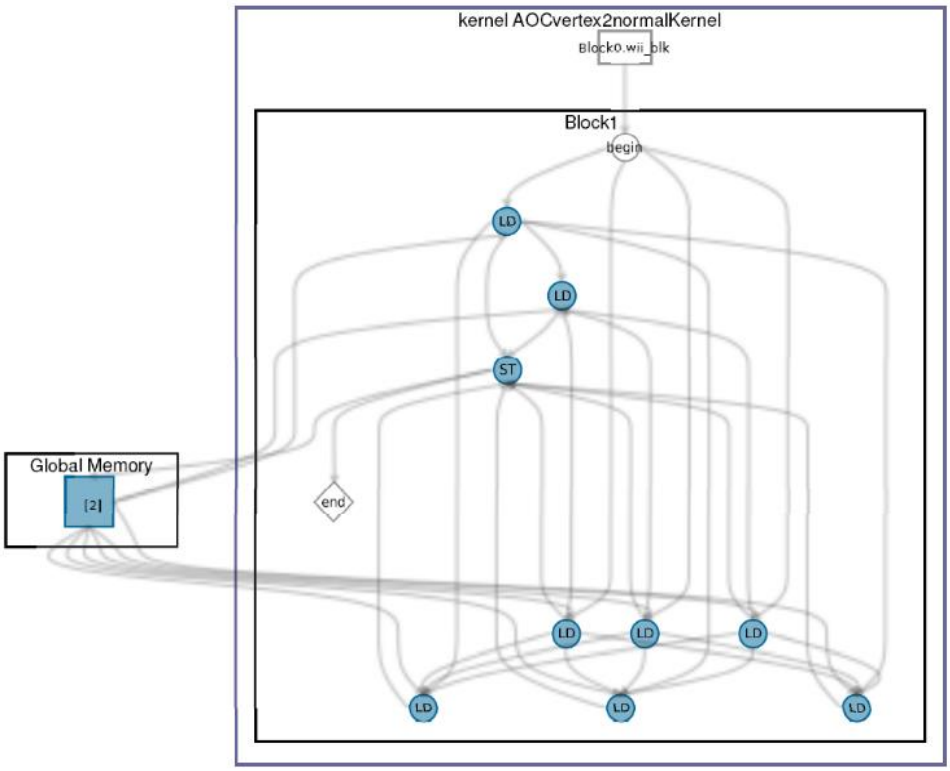


Figure 4.3.0: Vertex2Normal hardware diagram

Optimizations techniques used in this kernel are as follows:

- Asynchronous command queues
- DMA memory transfer
- loop pipelining
- Avoiding pointer aliasing
- Implementation of *find_max* and *find_min* function: These functions are not supported by Intel FPGA compiler and had to be explicitly implemented
- Loop unrolling

Optimized kernel can be found in Appendix. Although the loops are fully pipelined, there is a high number of simultaneous accesses to the global memory (as shown in figure 4.3.0), performing operations on left, right, up and down pixels. This has resulted in less speedup achievement for this kernel, even though the kernel is pipelined well.

Figure 4.3.1: FPGA unoptimized vs optimized depth2vertex execution time (ms)

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.322			0.322	6.096
execution time	0.523	0.083	0.533	1.021	0.240	0.292
read time		0.344			0.295	0.295

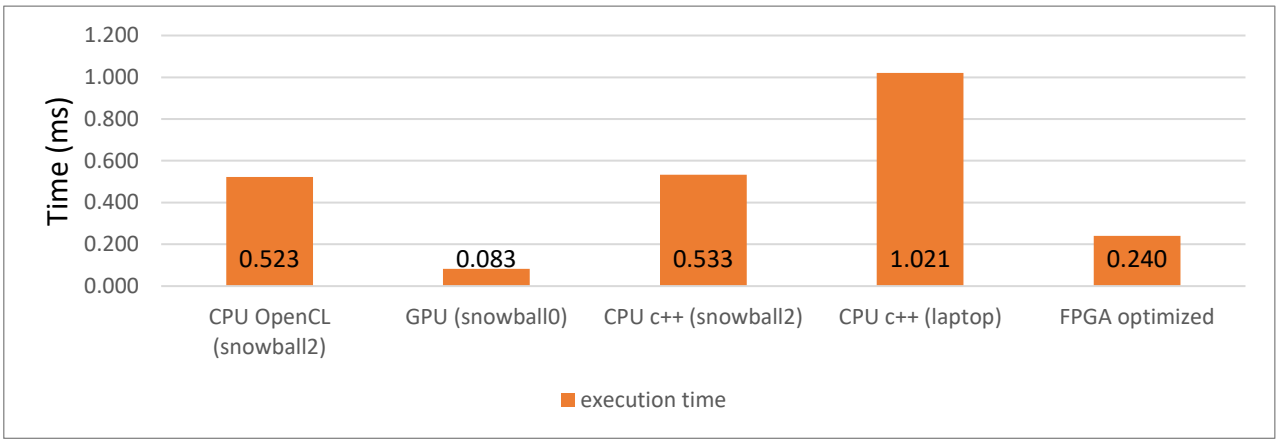


Figure 4.3.2: Vertex2Normal timings (ms) for FPGA optimized and all platforms

Analysis:

- The kernel is fully pipelined. Scope of optimization for this kernel is lower than the other kernels. The total processing time is 7.79 times faster. The execution time is 1.2x faster than the unoptimized FPGA execution time.
- What is also interesting to point out here is that the performance of CPU OpenCL is almost the same as CPU C++ on snowball2 machine. The reason for this is that, limiting the parallel core capability of CPU OpenCL platform.

- With regards to area, there is slight increase in the area due to explicit *min* and *max* functions implementations.

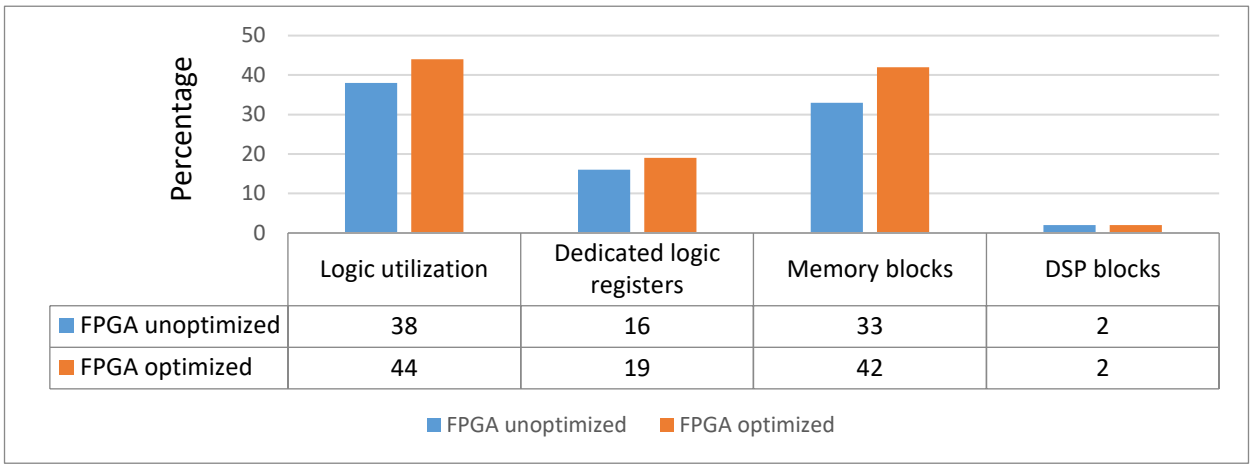


Figure 4.3.3: % of FPGA resource usage of vertex2normal optimized vs unoptimized

4.2.6 Track

This kernel establishes a correspondence between vertices in the synthetic and new point cloud [4].

```
__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);
  if(projPixel.x < 0 || projPixel.x > refVertexSize.x-1 || projPixel.y < 0 || projPixel.y > refVertexSize.y-1 ) {
    output[pixel.x + outputSize.x * pixel.y].result = -2;
    return;
  }
  const uint2 refPixel = (uint2) (projPixel.x, projPixel.y);
  const float3 referenceNormal = vload3(refPixel.x + refNormalSize.x * refPixel.y,refNormal);
  if(referenceNormal.x == INVALID) {
    output[pixel.x + outputSize.x * pixel.y].result = -3;
    return;
  }
  const float3 diff = vload3(refPixel.x + refVertexSize.x * refPixel.y,refVertex) - projectedVertex;
  const float3 projectedNormal = myrotate(Ttrack, inNormalPixel);
  if(length(diff) > dist_threshold ) {
    output[pixel.x + outputSize.x * pixel.y].result = -4;
    return;
  }
  if(dot(projectedNormal, referenceNormal) < normal_threshold) {
    output[pixel.x + outputSize.x * pixel.y] .result = -5;
    return;
  }
  output[pixel.x + outputSize.x * pixel.y].result = 1;
  output[pixel.x + outputSize.x * pixel.y].error = dot(referenceNormal, diff);
  vstore3(referenceNormal,0,(output[pixel.x + outputSize.x * pixel.y].J));
  vstore3(cross(projectedVertex, referenceNormal),1,(output[pixel.x + outputSize.x * pixel.y].J));
}
```

Figure 4.3.4: original track kernel

For this kernel, loop pipelining results the kernel results in high II, which cannot be resolved. However, due to high number of threads (320x240), it is possible to optimize this kernel using NDRange kernels strategies. Optimizations techniques used in this kernels are as follows:

- Avoiding pointer aliasing
- Kernel replication: only 2 compute units can be introduced for this kernel. The kernel occupies large area of the targeted FPGA and therefore, replication and vectorization was only possible with small factors.
- Kernel vectorization: Vectorization with 2 and 4 SIMD lanes results in 0.185 and 0.296 milliseconds execution time. Combination of replication and vectorization is not possible because the design does not fit on the FPGA.
- Defining work-group sizes: work group size of *(80,60,1)* ensures there are enough work groups to be executed in parallel on compute units.
- Asynchronous command queues.
- The best that could be done for loop-pipelined approach was to move the dependency to local memory. Nevertheless, because of high number of threads in this kernel (76800), increasing the number of compute units results in a better performance.
- From hardware point of view, block 0 is the conditional statement where loading from global emory takes place. Block 1 is where most of the computation take place.

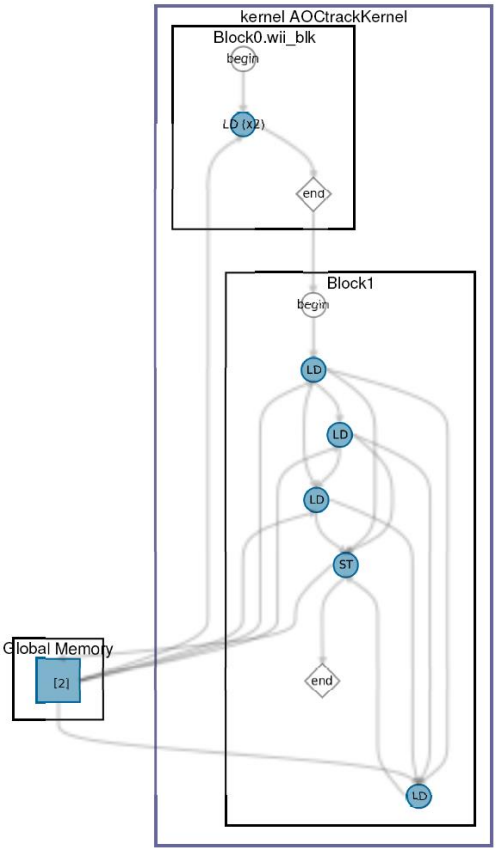


Figure 4.3.5: Vertex2Normal hardware diagram

```
363 __attribute__((reqd_work_group_size(80,60,1)))  //defining work group sizes
364 __attribute__(  (num_compute_units(2))  )       //defining 2 compute units
365 __kernel void AOCtrackKernel (
366     __global TrackData * restrict output,
367     const uint2 outputSize,
368     __global const float * restrict  inVertex,// float3
369     const uint2 inVertexSize,
370     __global const float * restrict inNormal,// float3
371     const uint2 inNormalSize,
372     __global const float * restrict refVertex,// float3
373     const uint2 refVertexSize,
374     __global const float * restrict refNormal,// float3
375     const uint2 refNormalSize,
376     __global const  Matrix4* restrict Ttrack,
377     __global const  Matrix4* restrict view,
378     const float dist_threshold,
379     const float normal_threshold,
380     const i
```

Figure 4.6: optimized section of the Track kernel

The kernel was therefore optimized and compiled as NDRange model with 2 compute units and a work-group size of *(80,60,1)*.

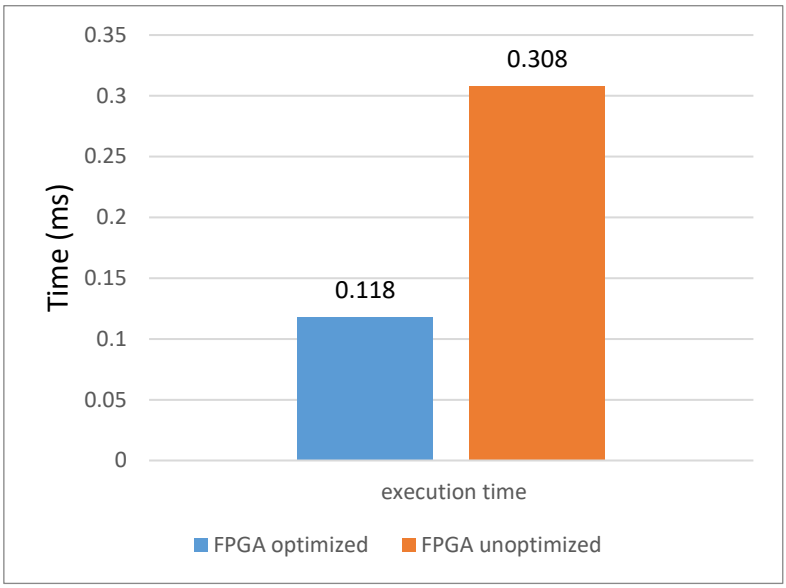


Figure 5.3.7: FPGA unoptimized vs optimized track execution time (ms)

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.162			0.392	1.687
execution time	0.107	0.0201	1.14	2.136	0.118	0.308
read time		0.494			0.906	315.754

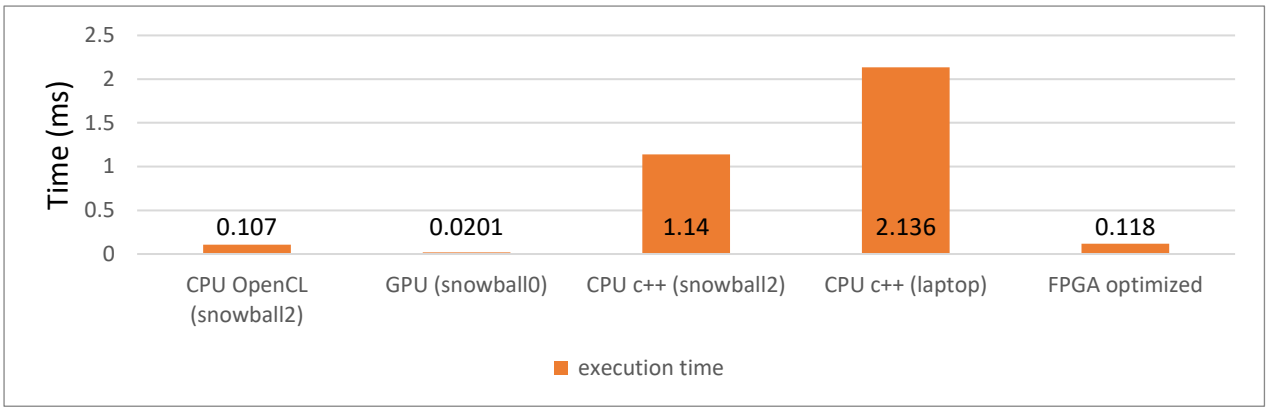


Figure 4.3.8: Track timings (ms) for FPGA optimized and all platforms

Analysis:

- The optimization has resulted in 2.6x speed un in execution time. The 64bit alignment has also reduced the read back time significantly from 315 ms to 0.906 ms., using DMA transfer.
- The important point about this function is its large resource usage of the FPGA, which have prevented further increasing the number of compute units and applying vectorization to the kernel. This is due to large structure arguments being stored in global memory, increasing the resources.

- The optimized FPGA execution time is 18x faster than the sequential CPU, reaching CPU OpenCL execution time.

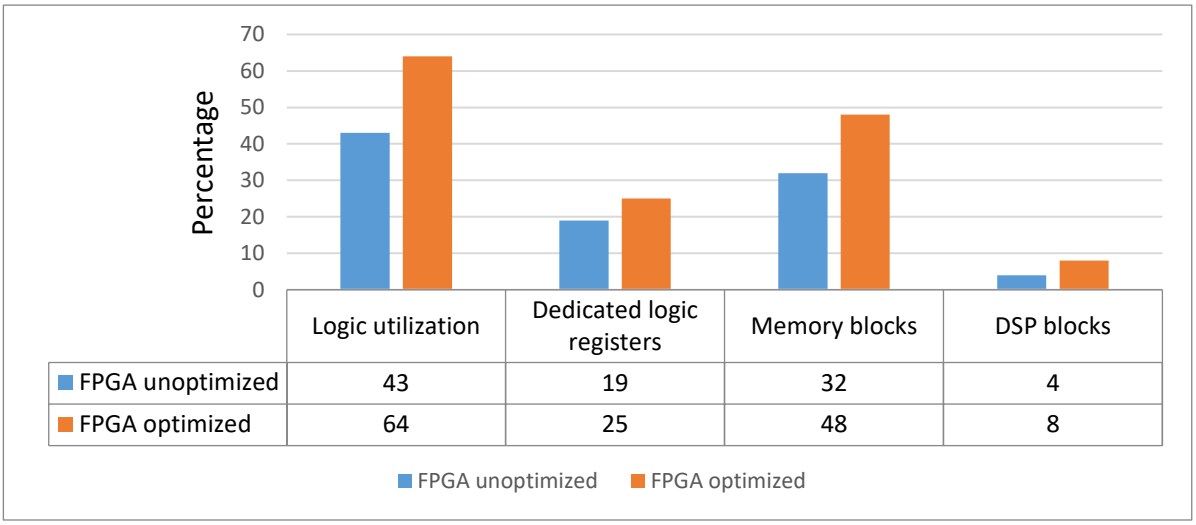


Figure 4.3.9: % of FPGA resource usage of track optimized vs unoptimized

4.2.7 Reduce

This kernel adds up all the errors of vertices for the minimization process. The full unoptimized kernel has been shown in Appendix for space saving. The kernel cannot be converted to single-work-item due to the barriers in the kernel.

Optimizations strategies used in this kernel are:

- Avoiding pointer aliasing
- Asynchronous command queues
- Pragma unrolling.
- Vectorization (*num_simd_work_items()*) is not possible because branching in the kernel is ID dependent. That is, in other words, the number of iterations of the nested for-loop depends on the work-item id(*blockIdx* and *threadIdx*) as shown in figure 4.4.0.

```
uint blockIdx = get_group_id(0);
uint blockDim = get_local_size(0);
uint threadIdx = get_local_id(0);
uint gridDim = get_num_groups(0);

const uint sline = threadIdx;

float sums[32];
float * jtj = sums + 7;
float * info = sums + 28;

for(uint i = 0; i < 32; ++i)
sums[i] = 0.0f;

for(uint y = blockIdx; y < size.y; y += gridDim) {
  for(uint x = sline; x < size.x; x += blockDim ) {
    const TrackData row = J[x + y * JSize.x];
    if(row.result < 1) {
      info[1] += row.result == -4 ? 1 : 0;
      info[2] += row.result == -5 ? 1 : 0;
      info[3] += row.result > -4 ? 1 : 0;
      continue;
    }
```

Figure 4.4.0: part of the reduce kernel preventing vectorization

- Replication: the optimized kernel is executed with 2 compute units. More than two compute units makes the kernel not fit on the FPGA device.
- Specifying required work group size: A work-group size of *(64,1,1)* is introduced. (The iteration space of this kernel is one dimensional)
- As shown in in hardware diagram (figure 4.4.2) , there exists loop-dependencies in many parts of the kernel (variable *jtj* and *info*), preventing loop-pipelining the kernel.

Full optimized code can also be found in the Appendix. The final optimized kernel was compiled in NDRange with 2 compute units.

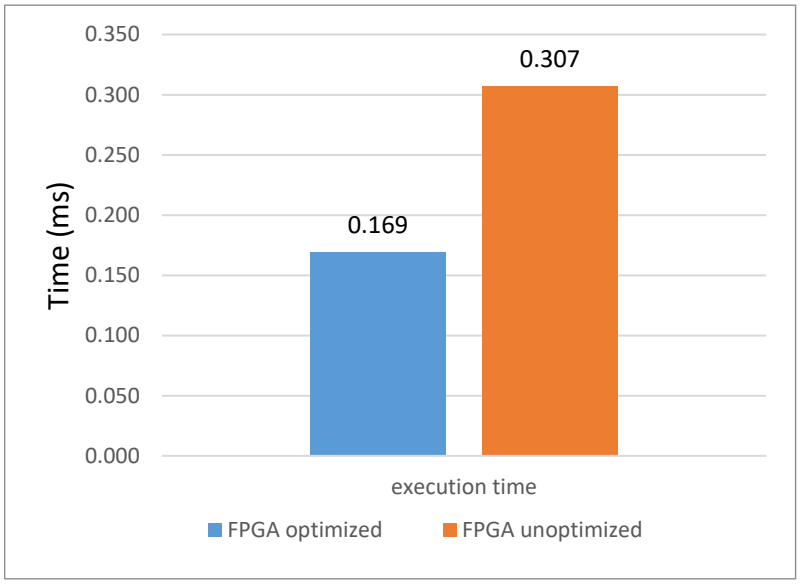


Figure 4.4.1: FPGA unoptimized vs optimized reduce execution time (ms)

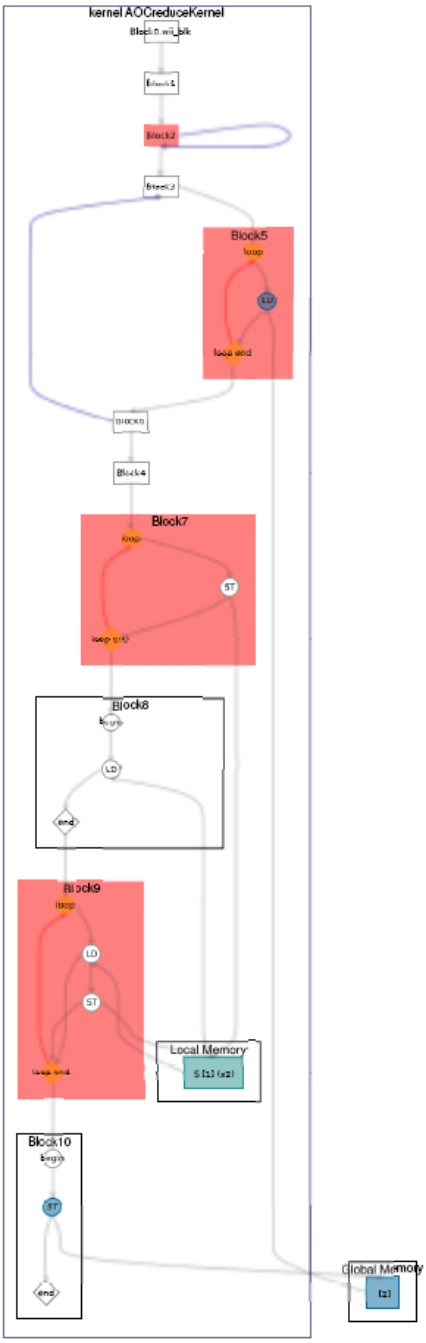


Figure 4.4.2: Reduce hardware diagram

Timing (ms)	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.751			1.137	27.780
execution time	0.475	0.123	0.296	1.412	0.169	0.307
read time		0.002			0.057	0.054

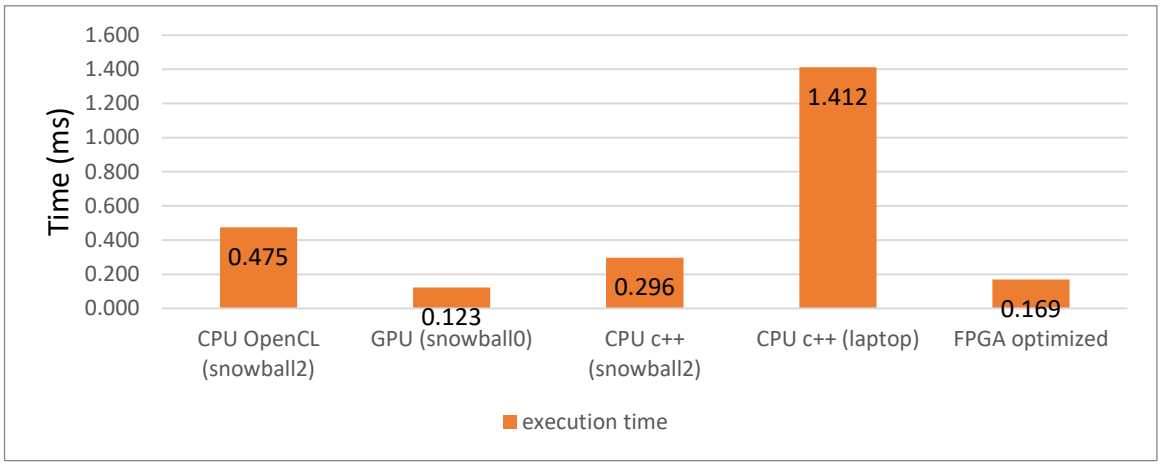


Figure 4.4.3: Reduce timings (ms) for FPGA optimized and all platforms

The total kernel time on the FPGA is scientifically faster than the unoptimized version (20x speedup). Kernel execution time is 1.8 times faster on FPGA optimized that unoptimized version.

- Further optimization is not possible because of lack of resources to increase vectorization and replication in the kernel.
- The FPGA optimized execution time is 4x faster than CPU OpenCL and is very close to the GPU execution time.
- This kernel by default, is resource intensive kernel because of large floating point arithmetic taking place in the kernel.

As before the area analysis of this kernel is:

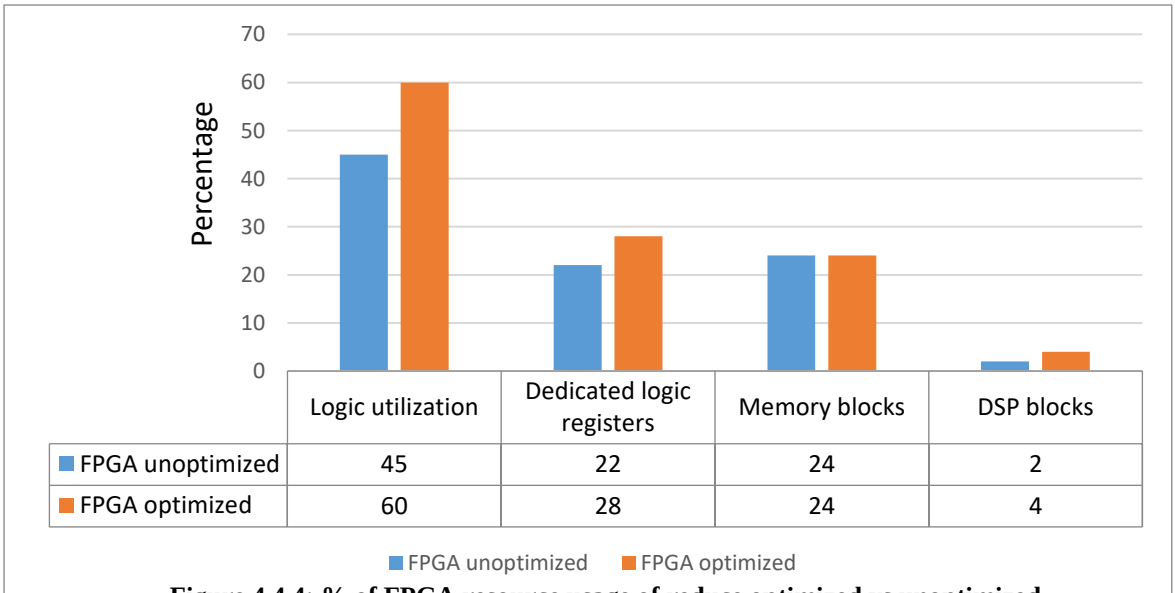
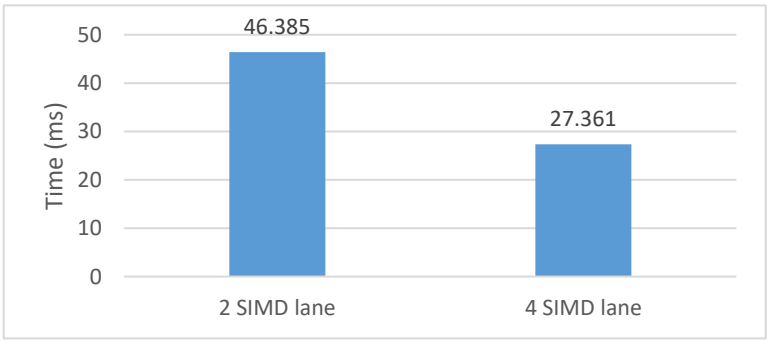


Figure 4.4.4: % of FPGA resource usage of reduce optimized vs unoptimized

4.2.8 Integrate

This kernel integrates the new point cloud in the 3D volume. As we move towards the final kernels, the complexity of kernels increase and therefore, for saving space, the kernel can be found in Appendix. Converting to single-work-item resulted in loop dependencies that cannot be resolved as shown in the hardware diagram. Therefore, optimizations strategies used in this kernel are:

- Passing Structures data type as arguments: Again, a pointer to *Matrix4* structure have been passed for *invTrack* and *K* arguments
- DMA transfer
- Avoiding pointer aliasing.
- Specifying a work-group-size: work group size of *(32,32,1)* ensures there are enough compute units to handle are all the work-groups in parallel
- Vectorization: This has been measured with 2 and 4 SIMD lanes resulting in execution time of 46.385 and 27.361 ms execution time respectively.



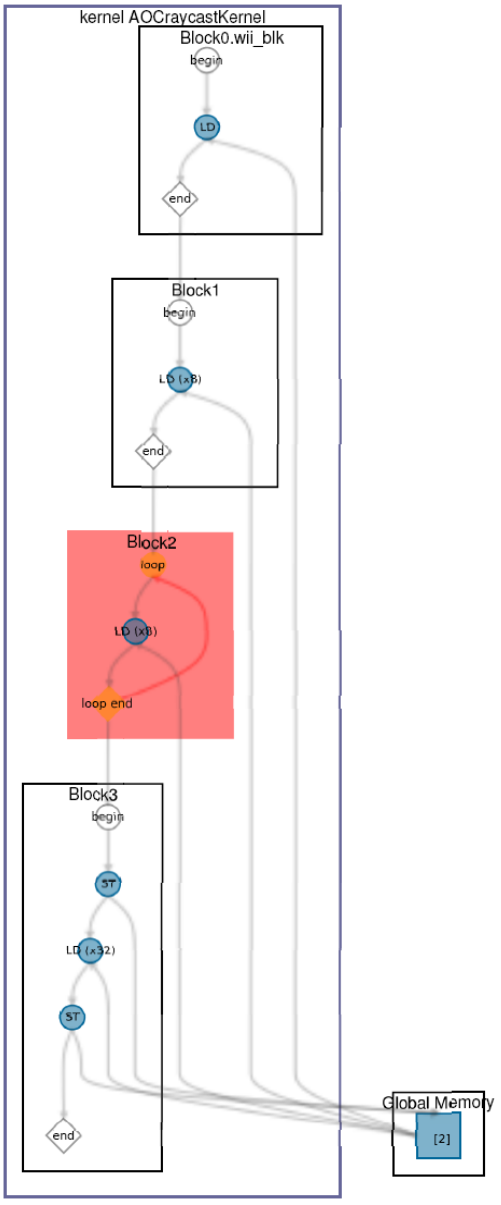
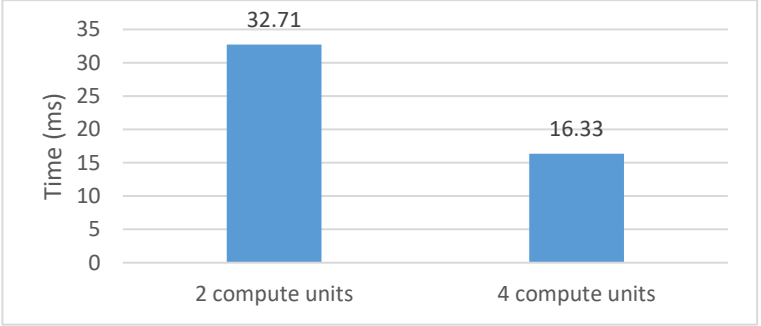


Figure 4.4.5: Integrate hardware diagram

- Replication: Replication has been measured with 2 and 4 compute units resulting in 32.71ms and 16.433ms execution time respectively.



The kernel is therefore compiled as NDRange and the best match (2 compute units with specified work group size) has been chosen for optimized measurements.

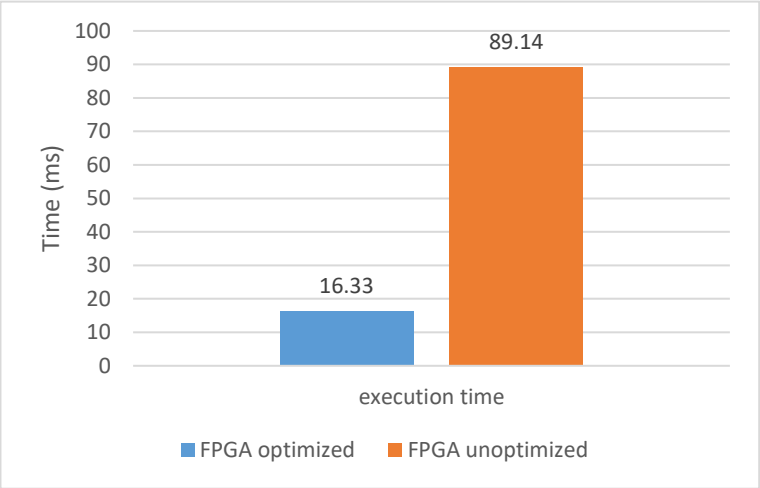


Figure 4.4.6: FPGA unoptimized vs optimized kernel execution time (ms)

Timing	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.1			0.291	0.289
execution time	29.507	3.565	170.676	227.099	16.33	89.14
read time		20.112			38.11	9426.6

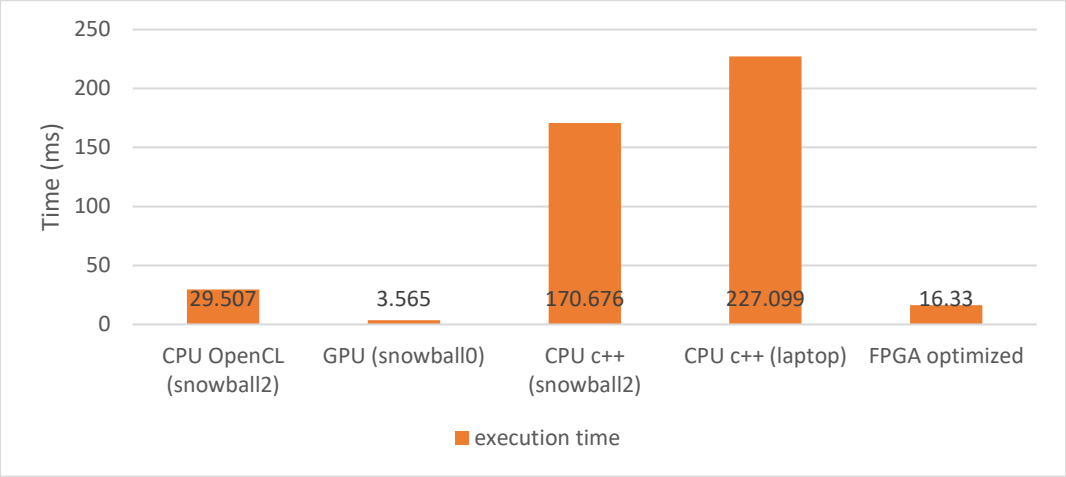


Figure 4.4.7: Integrate timings (ms) for FPGA optimized and all other platforms

- This is a computationally expensive kernel where the total kernel time on the FPGA is scientifically faster than the unoptimized version, especially because of large read time of unoptimized version due to lack of alignment, (execution time is 5.4 times faster).
- With the optimization applied, the FPGA execution time is 1.8x faster than the multi core CPU and 13.9 times faster than the laptop CPU.

- More replication is not possible since further optimization is resource bounded as shown in figure 4.4.7.

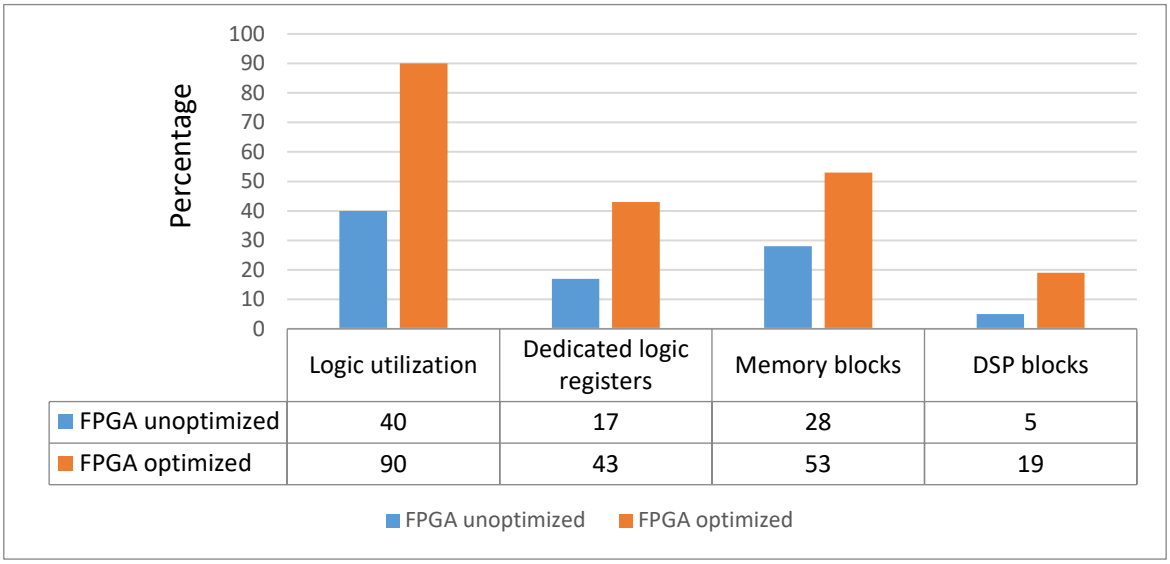


Figure 4.4.8: % of FPGA area usage of integrate optimized vs unoptimized

4.2.9 Raycast

This kernel computes the point cloud and normal of the current estimate of the camera position [4]. This is a complex kernel which can be found in Appendix.

Since the kernel occupies a large area on the FPGA, none of the kernel execution optimizations could be applied.

Therefore, the only optimizations strategies used are:

- Asynchronous command queues
- DMA transfer
- Avoiding pointer aliasing

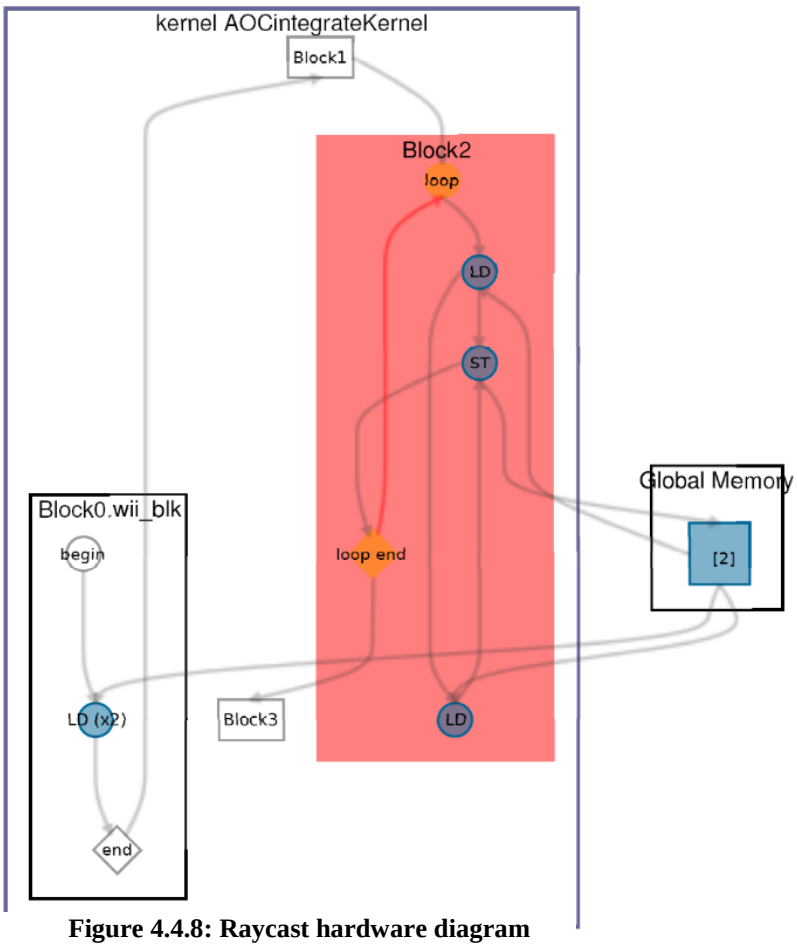


Figure 4.4.8: Raycast hardware diagram

Timing	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		20.551			29.044	29.291
execution time	41.875	3.299	185.127	237.596	127.058	127.058
read time		6.892			8.203	51.57

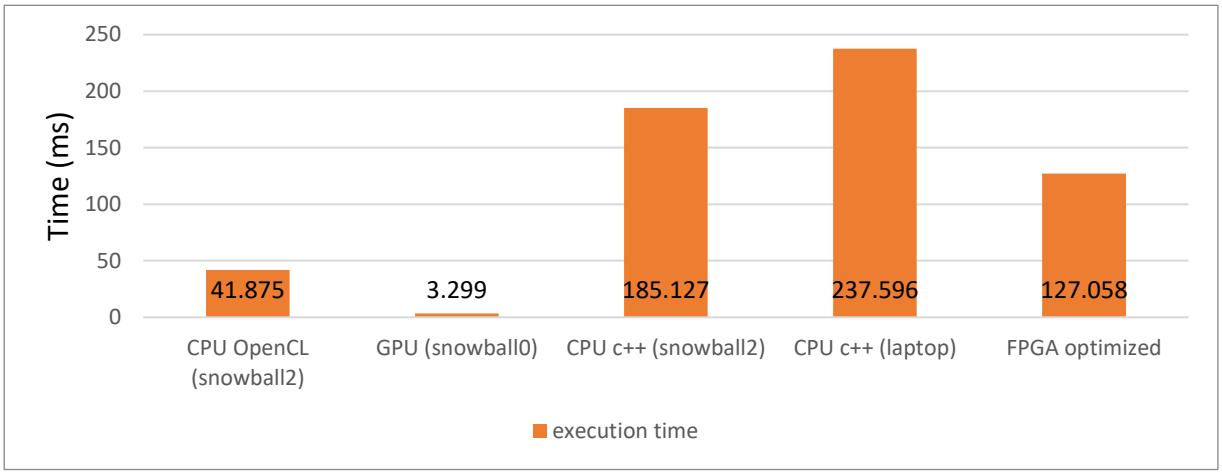


Figure 4.4.9: Raycast timings (ms) for FPGA optimized and all other platforms

- The main problem is that we have memory accesses to a volume, which depend on the actual position (and view) which are not known during compile time. Such functions are better suited for GPUs, as they can render each point independently and at the same time hide the memory latencies.
- For this kernel, the FPGA just outperforms the sequential CPU, and due to the reason mentioned above, it behaves worse than the CPU OpenCL and GPU.

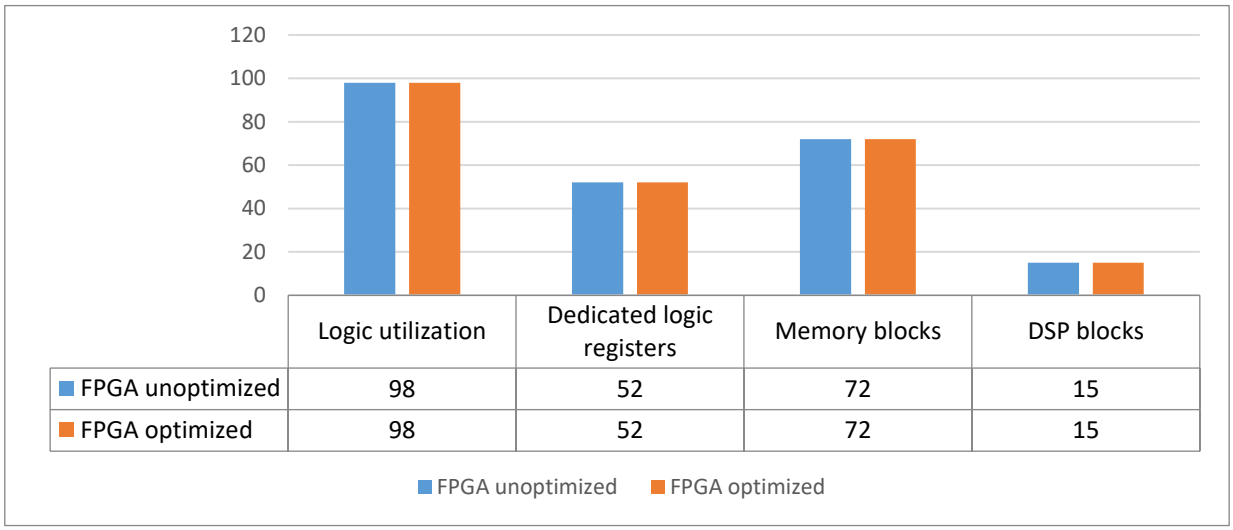


Figure 4.5.0: % of FPGA area usage of raycast optimized vs unoptimized

Rendering kernels

Same approach as above has taken for the rendering functions (*renderdepth*, *rendertrack* and *rendervolume*) and the final last results are just shown. The optimized and unoptimized code of rendering functions can be found in appendix.

Render Depth: This kernel visualises the depth map with colour coding.

Timing	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.1			0.239	0.276
execution time	0.079	0.0214	0.492	0.854	0.189	0.41
read time		0.0989			0.203	43.467

Figure 4.5.1: Render depth timings (ms) for FPGA optimized and all platforms

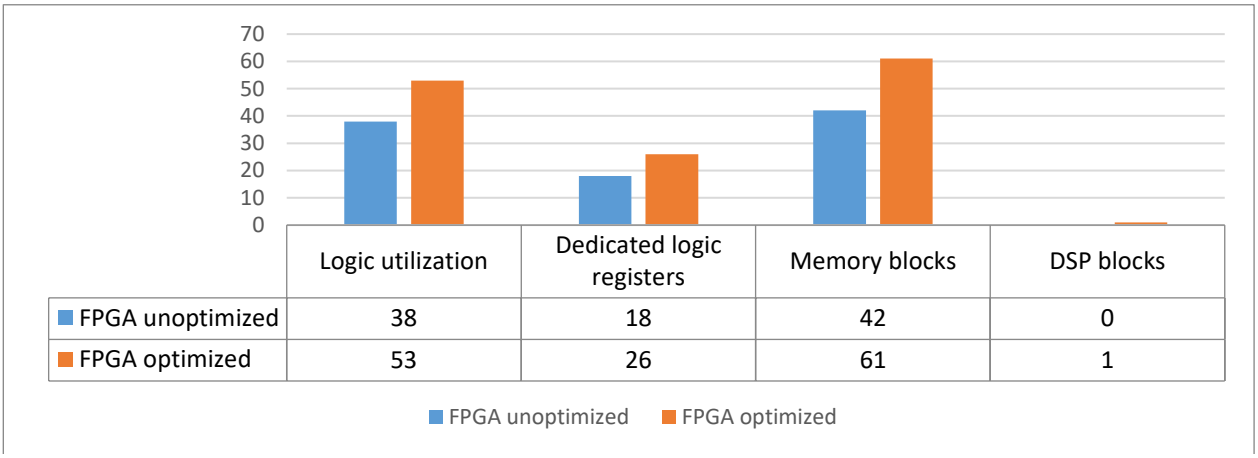


Figure 4.5.2: % of FPGA area usage of render depth optimized vs unoptimized

Render Track: As appears from its name, this kernel visualises the tracking result.

Timing	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		0.749			1.187	27.771
execution time	0.341	0.0476	0.417	0.527	0.221	0.449
read time		0.0992			0.189	44.102

Figure 4.7: Render track timings (ms) for FPGA optimized and all platforms

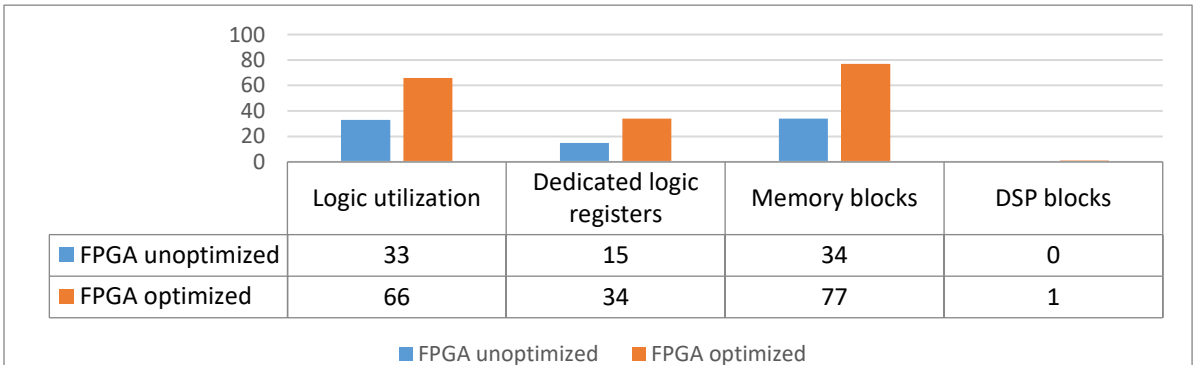
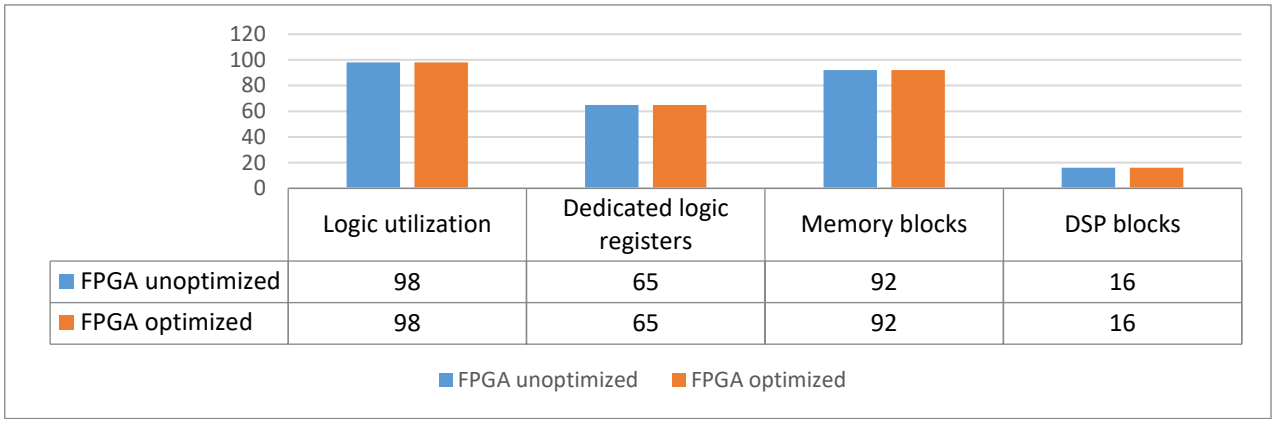


Figure 4.5.4: % of FPGA area usage of render track optimized vs unoptimized

Render Volume: This kernel visualises the 3D reconstruction from a fixed viewpoint.

Timing	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
write time		20.55			28.988	30.193
execution time	40.666	3.262	165.372	278.624	124.334	124.334
read time		0.1			0.182	44.33

Figure 4.5.5: Render volume timings (ms) for FPGA optimized and all platforms



	Logic utilization	Dedicated logic registers	Memory blocks	DSP blocks
FPGA unoptimized	98	65	92	16
FPGA optimized	98	65	92	16

Figure 4.5.6: % of FPGA area usage of render volume optimized vs unoptimized

Analysis of all rendering kernels:

- The structure of render volume kernel is just like the *raycast* kernel. Therefore, none of the kernel optimizations could be applied to it apart from transfer time optimizations. As mentioned before, these types of kernels are best fitted on GPUs, which executes the kernel in 3.62 milliseconds only.
- All rendering kernels are executed are compiled and optimized in NDRange.
- No further increase in replication and vectorization factor could be introduced for render depth and render track kernels, due to resources constraints.
- Optimizations applied to *render depth* and *render track* has made the execution time of this kernel over 2x faster.

4.3 Evaluation

Every single kernel in the framework has been analysed, optimized for an FPGA and their performance has been evaluated on different platforms with regards to execution time, FPGA resource usage and FPGA accuracy (ATE). The overall picture of per-kernel evaluation is as follows:

1. **Time:**

The optimizations applied to the per-kernel evaluation of timing have been on total time, which is composed of write, kernel execution and read back time. Table below summarizes the FPGA total timings (ms) comparison:

Kernel	Total FPGA optimized (ms)	Total FPGA unoptimized (ms)	Speedup (unoptimized/optimized)
mm2	0.874	7.673	8.779
Bilateral Filter	1.420	51.793	36.474
Half sample	0.460	3.849	8.376
Depth2vertex	0.510	76.704	150.341
Vertex2Normal	0.857	6.683	7.798
Track	1.416	317.749	224.399
Reduce	1.363	28.141	20.646
Integrate	54.731	9516.029	173.869
Raycast	164.305	207.919	1.265
Render depth	0.631	44.153	69.973
Render Track	1.597	72.322	45.286
Render Volume	153.504	198.857	1.295
Average	**31.806**	**877.656**	**62.375**

Table 4.6.0: Total FPGA optimized vs unoptimized timings (ms) for per-kernel evaluation

- As can be seen in table 4.6, optimizations for **per-kernel evaluation** has resulted an average of **62x** speed up on the FPGA device, compared to the original SLAMBench implementation with regards to **total time**.
- The best accelerations are achieved for *depth2vertex*, *track* and *integrate* kernels where they involve many memory transfers between the host CPU and the FPGA device. This is due to the Direct Memory Access (64bit alignment) having more effect for these types of kernels.
- For *Raycast* and *RenderVolume* kernels, no optimization could be applied at kernel-level. Thus, the only optimizations applied are with regards to transfer time and therefore, minimum speed up is achieved for them.
- The most time-consuming kernels on the FPGA device are *Raycast*, *RenderVolume* and *integrate* kernels. These kernels have a complex structure, with many structure data types, additional functions implementations and heavy floating point arithmetic, occupying large resources on the FPGA device.

A better comparison with regards to timing would be analysing just the execution time of each kernel on the FPGA device, removing the transfer time created for per-kernel evaluation. This evaluates the kernel-level optimizations applied.

Kernel	FPGA optimized (ms)	FPGA unoptimized (ms)	speedup	approach	Optimized % of time
mm2	0.153	0.341	2.229	single-work-item	0.057
Bilateral Filter	0.695	7.526	10.829	single-work-item	0.258
Half sample	0.089	0.222	2.508	single-work-item	0.033
Depth2vertex	0.079	0.413	5.215	single-work-item	0.029
Vertex2Normal	0.240	0.292	1.217	single-work-item	0.089
Track	0.118	0.308	2.610	NDRange	0.044
Reduce	0.169	0.307	1.817	NDRange	0.063
Integrate	16.330	89.140	5.459	NDRange	6.055
Raycast	127.058	127.058	1.000	NDRange	47.115
Render depth	0.189	0.410	2.169	NDRange	0.070
Render Track	0.221	0.449	2.032	NDRange	0.082
RenderVolume	124.334	124.334	1.000	NDRange	46.105
Average	**22.473**	**29.233**	**3.174**		

Table 4.6.1: Kernel execution FPGA optimized vs unoptimized timings (ms) for per-kernel evaluation

- The optimization at kernel-level has resulted an average speed up of **3.1x** compared to the unoptimized version.
- The First 5 kernels have been optimized in single-work-item strategies and resulted in fully pipelined loops in all parts of the kernel. Changing to single-work-item have not been possible for the rest of the kernels (due to unresolved dependencies that result in very high initiation interval) and therefore, NDRange optimization approach have been applied to them instead.
- Both approaches have resulted in speed ups. In single work-item model, for kernels that degree of optimization was high (best map to the pipeline architecture) more speedup has been achieved (*bilateral filter* and *depth2vertex*)
- As mentioned before, due to memory accesses that depend on parameters that are not known at compile time, no speedup was obtained for *raycast* and *renderVolume* kernel.
- For the NDRange kernels, no further optimizations (replication, vectorization) could be applied due to the limited resources availability of the FPGA. For single-work-item kernels the status of all the loops are all fully pipelined and therefore no further pipeline papalism optimization could be applied further.
- *Integrate* kernel has seen the highest speedup in NDRange model, due to high number of work-items (threads=76800) available in this kernel.
- For kernels that do not contains loops and have high number of global memory accesses, degree of optimization is low and the amount of speed up is lower than the others (*Vertex2normal*, *reduce*, *Render track*).
- percentage of optimized kernel execution time spent on *Raycast* and *RenderVolume* (47 and 46%) is the highest, where no optimizations could be applied. This is

followed by the integrate kernel occupying 6% of total execution time. These three kernels occupy 99% of KinectFusion execution time on the FPGA device..

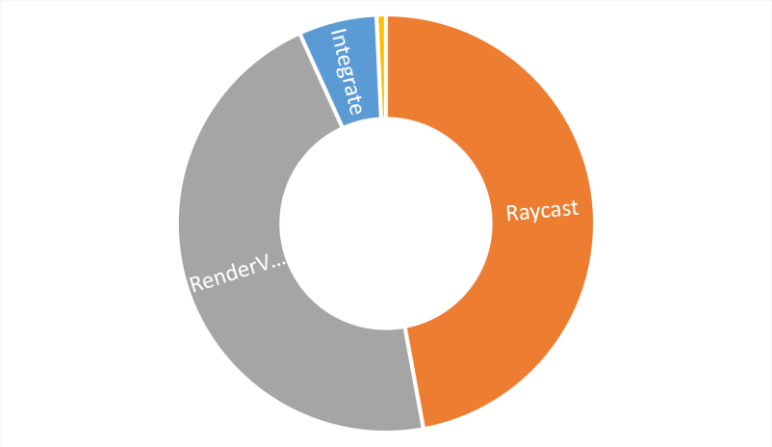


Figure 4.6.2: proportion of FPGA optimized execution time in each kernel

Comparison between FPGA and all other platforms execution time (ms) for each kernel is as follows:

Kernel	CPU OpenCL (snowball2)	GPU (snowball0)	CPU C++ (snowball2)	CPU C++ (laptop)	FPGA optimized	FPGA unoptimized
mm2	0.145	0.054	0.465	0.370	0.153	0.341
Bilateral Filter	5.213	1.002	44.462	75.283	0.695	7.526
Half sample	0.0654	0.03435	0.106	0.242	0.0885	0.222
Depth2vertex	0.105	0.0115	0.205	0.427	0.0792	0.413
Vertex2Normal	0.523	0.083	0.533	1.021	0.240	0.292
Track	0.107	0.0201	1.14	2.136	0.118	0.308
Reduce	0.475	0.123	0.296	1.412	0.169	0.307
Integrate	29.507	3.565	170.676	227.099	16.33	89.14
Raycast	41.875	3.299	185.127	237.596	127.058	127.058
Render depth	0.079	0.0214	0.492	0.854	0.189	0.41
Render Track	0.341	0.0476	0.417	0.527	0.221	0.449
Render Volume	40.666	3.262	165.372	278.624	124.334	124.334

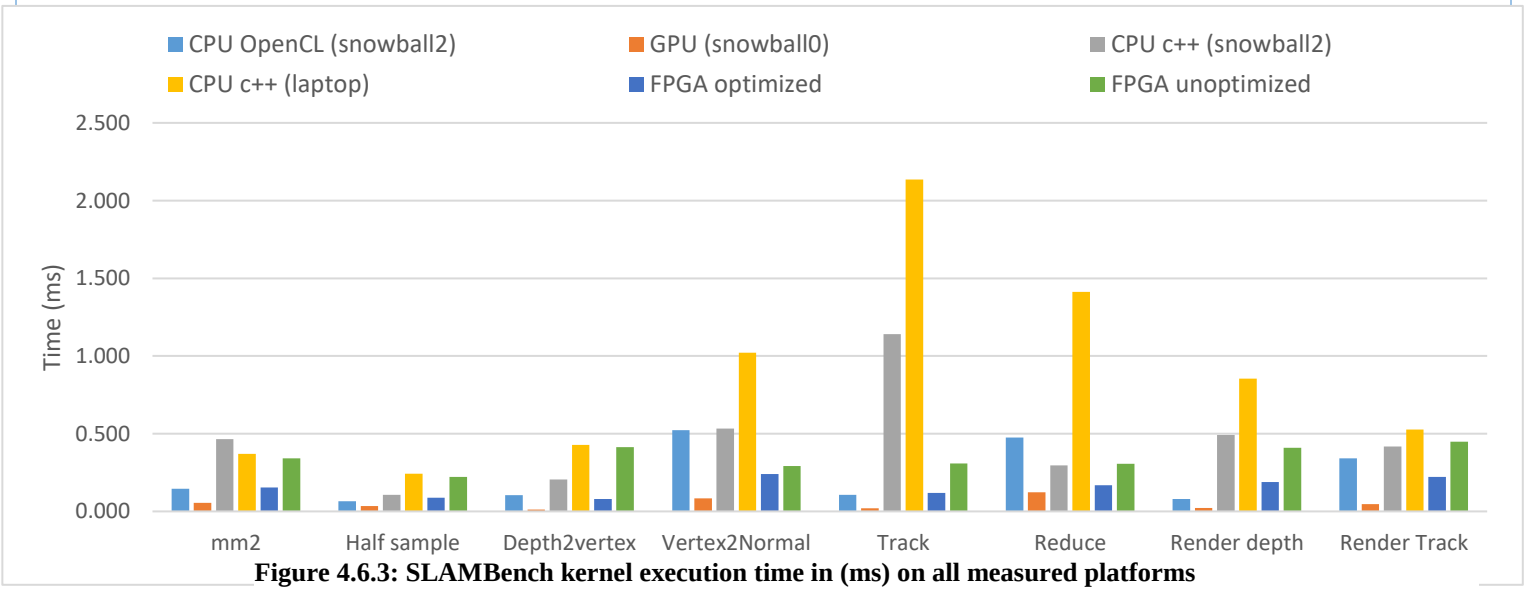


Figure 4.6.3: SLAMBench kernel execution time in (ms) on all measured platforms

- As can be seen in all kernels, the sequential C++ CPU performance is the worst by far amongst all the platforms (*laptop* and *snowball2*), which is acceptable as no parallelism is introduced in their execution.
- This is followed by the FPGA unoptimized platform for many kernels. This is again acceptable considering no optimization is made to tune the OpenCL kernel for the FPGA device.
- Next platform is Multi-core CPU (CPU OpenCL), which has on average 2x execution time as the unoptimized FPGA.
- The FPGA optimization strategies have made FPGA optimized version faster than multi-core CPU for most of the kernels. This is what was expected, since with correct algorithm design, FPGAs can beat modern multi-core CPUs.
- Finally, the fastest version on most of the kernels is GPU. SLAM algorithms like KinectFusion has very large number of off chip memory accesses and parameters like (*position*, *view*, *input scene*) that are not known during kernel compilation time on FPGAs. This gives GPUs advantage to other platforms, as the OpenCL implementation best matches to its architecture and many memory latencies can be hidden. (More details are explained in final evaluation section).

2. ATE:

Table below (4.6.4) summarizes the ATE of all kernels for the FPGA device.

Kernel	FPGA unoptimized	FPGA optimized
mm2	0.0206	0.0206
Bilateral Filter	0.0201	0.0212
Half sample	0.0224	0.0205
Depth2vertex	0.0212	0.0205
Vertex2Normal	0.0409	0.049
Track	0.0212	0.0232
Reduce	0.0205	0.0209
Integrate	0.0212	0.0213
Raycast	0.0196	0.0196
Render depth	0.0230	0.0230
Render Track	0.0209	0.0225
Render Volume	0.0187	0.0187

Table 4.6.4: ATE (meters) for optimized vs unoptimized per-kernel evaluation

The ATE of all the kernels before and after optimizations are all acceptable in a range of a few centimetres. The precision of floating point numbers has not been changed in any of the kernels. The same floating point arithmetic results on different OpenCL devices are not the same, and

there exists a very small difference between different hardware platforms, because of the order of doing the floating-point arithmetic. (more details in section 4.4).

3. **Resource usage:**

As mentioned before, there is a trade-off between the FPGA execution time and resource usage in the analysis. For all-kernels the optimizations resulted in an increase in the FPGA resource usage (mostly logic utilization and memory blocks). The optimizations for NDRange kernels have continued until it takes advantage of all available area usage on the targeted FPGA device.

What is evident, is that it is not possible to fit all the kernels on the FPGA device, even in unoptimized format. Thus, the next optimization is aimed to reduce the area of the kernels to come up with the best model for the FPGA platform in this framework.

4.4 Optimizing floating point operations

FPGAs contain major logic for implementing floating point operations. It is possible to manually set Intel FPGA SDK Compiler to perform optimizations that create more efficient pipeline structure and reduce the hardware usage of the kernels, yet making small differences in the floating-point results [30]. This optimization has been analysed for to the most time consuming optimized kernels, which are *raycast*, *integrate*, *rendervolume* and *reduce*.

Tree balancing:

Order of floating point operations do matter in OpenCL. By default, the compiler creates non-balanced tree floating point arithmetic, which is long and expensive in hardware. A more efficient hardware implementation is when the a balanced-tree hardware is generated for the operation. As an example, considering the following arithmetic [30]:

result = (((A x B) + C) + (D x E)) + (F x G)

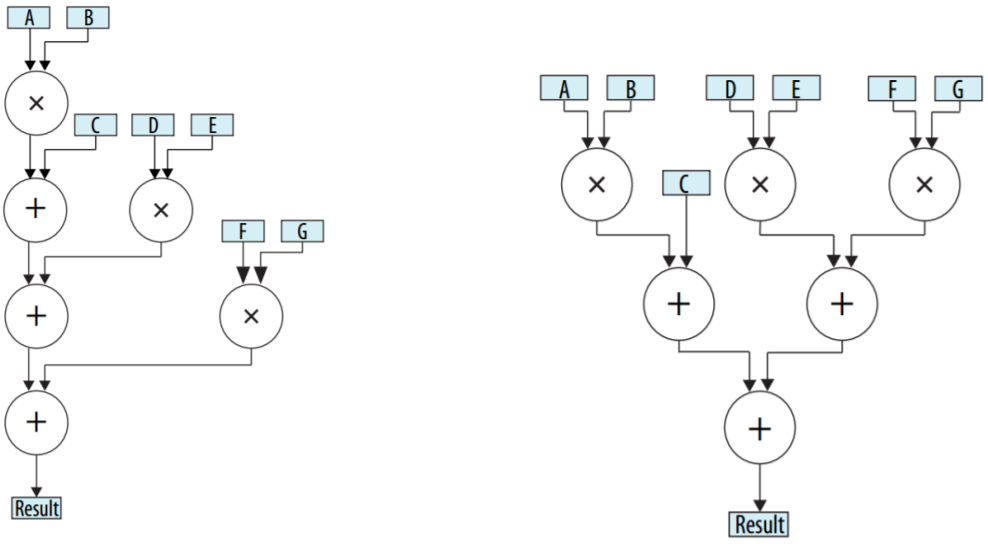


Figure 4.6.5: Balanced vs non-balanced tree floating point hardware implementation [30]

This optimization does not happen by default, because the result of balance-tree floating point is not consistent with IEEE 754-2008 standard. What is being analysed here, is how SLAM ATE differ once this optimization is applied, and whether small differences in mentioned kernels result is tolerable in the overall SLAM application.

What was seen is that overall ATE of SLAM kernels (what has been reported so far, which is the total average ATE of whole application) has not changed. This means that the overall SLAM application run successfully without losing the camera. However, specific ATE corresponding to each kernel has changed, since only the targeted kernel is running on the FPGA device.

Table below summarizes the how kernel-specific ATE (meters) has been affected.

kernel	unbalanced kernel-specific ATE (m)	balanced kernel-specific ATE (m)
raycast	0.183	0.201
reduce	0.0263	0.286
render volume	0.0315	0.155
vertex2normal	0.0414	0.042
integrate	0.163	0.171

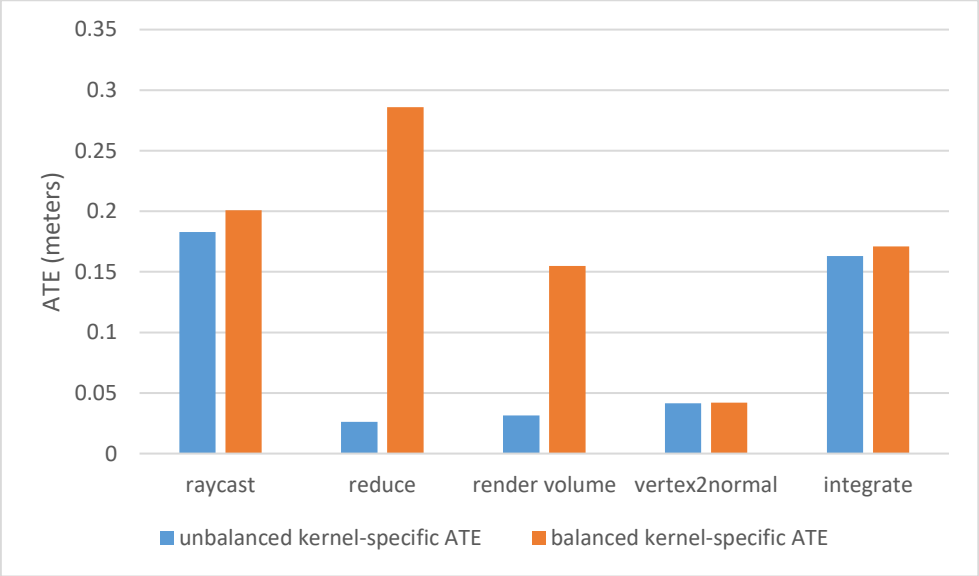


Figure 4.6.5:6 : kernels ATE (meters) before and after floating point optimization

What can be seen is that there is an increase in ATE of specific kernels, as expected. This is more evident for floating-point intensive kernels (*reduce* and *renderVolume*), where the ATE has exceeded the acceptable range of 4.5 centimetres.

5 Results

After analysing every single kernel and applying optimizations to every single of them, this section groups the kernels together, and apply optimization to a set of them.

5.1 putting kernels together

It is important to mention here that the host-kernel transfer time is introduced only if some kernels are executed on the host CPU and some on the kernel device (e.g. in per-kernel evaluation). If all kernels were executed in OpenCL format, like original SLAMBench OpenCL KinectFusion implementation, there would only be one write to the OpenCL device at the beginning, and one read back from the OpenCL device at the end of each frame computation (all variables could be stored in their OpenCL format).

What is really evident, is that for the targeted FPGA, it is still not possible to fit all the optimized kernels to the FPGA device. To evaluate the final FPGA OpenCL design, the optimized FPGA kernels have been grouped together, fitting as much as possible on the FPGA in each group. It is worth mentioning here that, the grouped kernels have been compiled with *(--high-effort)* flag, to force the compiler fit the FPGA device as much as it can.

As a result, the whole architecture can be summarized in the diagram below:

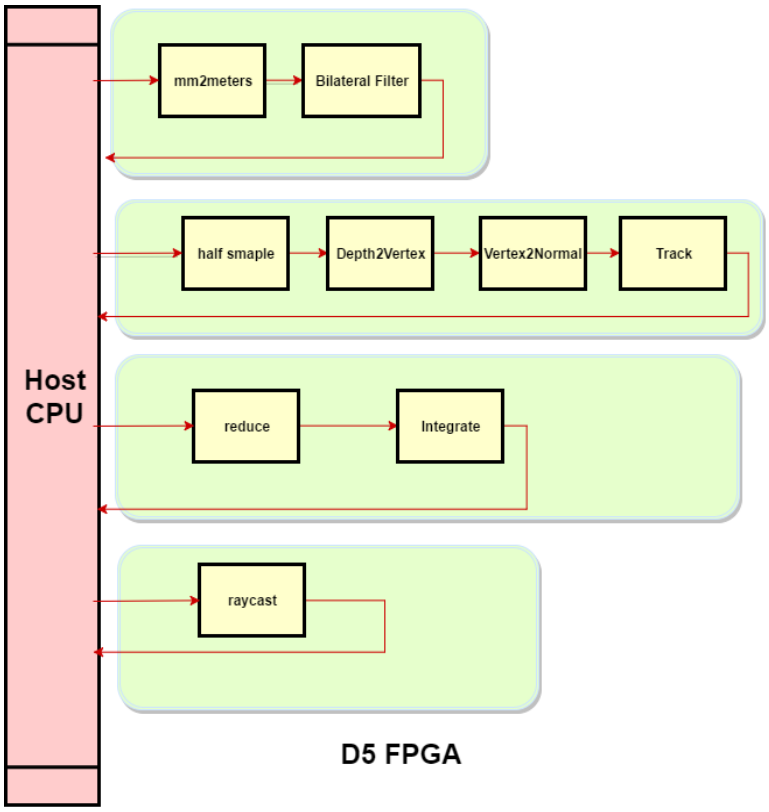


Figure 5.1: grouped kernels

As can be seen, if all kernels are intended to be executed on the FPGA, the optimized kernels should be divided in to 4 groups:

1. Group 1: *mm2meters* and *bilateral filter* kernels.
2. Group 2: *Half sample, Depth2Vertex, Vertex2Normal* and *Track* kernels.
3. Group 3: *Reduce* and *Integrate* kernels.
4. Group 4: *Raycast*.

The rendering kernels have all been off loaded to the host CPU because they are not included in the SLAMBench platform evaluation (*process_frame()* function provided by SLAMBench). As mentioned before, for consecutive kernels, there is no need to write the data back and forth every time, and only one write and read transfer is needed in the beginning and the end of each group.

Therefore, the FPGA needs to be reprogrammed 4 times for each frame in the scene (calling four different .aocx files one after another for each frame). Given that the FPGA reprogramming time takes a large amount of time (~1823ms), the FPGA spends most of its time in programming state. This results in d5 FPGA OpenCL, executing the whole application in average of 7.440 ms per frame (0.13 frames per second).

It is clear that this performance is not ideal, and due to the resources constraints of the Stratix V d5 FPGA, reprogramming must be avoided. The evaluation section at the end of the report does a more accurate evaluation, predicting and comparing what the performance would be if all the kernels could fit on the D5 FPGA For now, it can be deduced that, the targeted FPGA is best for targeting **single-kernel acceleration** in SLAMBench framework. The question of which kernels should be mapped to the FPGA device and what design yields the highest performance is fully discussed in “Best Match” section later in the report.

5.2 Channels

As noticed, the FPGA communication to the host CPU is one of the performance bottlenecks of the FPGA application. Even with Direct Memory Access (DMA) transfer, the transfer time between the host and the FPGA is twice compared to the GPU and should be avoided if possible. A unique concept about FPGAs is the concept of Intel FPGA SDK channels. In this method, kernels can communicate directly with each other with FIFO channels, without accessing global memory, increasing the data communication efficiency between the kernels. Figure below describes the memory pattern before and after channels implementation.

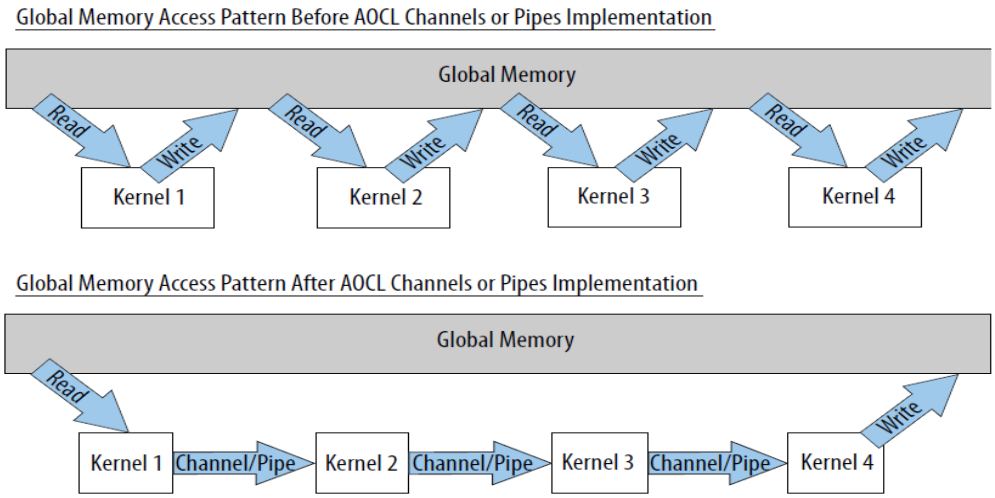


Figure 5.2: memory pattern of channels/pipes implementation [30]

Analysing the KinectFusion algorithm thoroughly, opportunities that these channels can be implemented is analysed (channels can only be implemented for kernels that are in the same group (1-4) specified before).

1. Group1 (Mm2meters-Bilateral Filter): A channel with correct depth (mm2meters output) has been created. The channel is filled at the end of mm2meters kernel and then gets read at the beginning of bilateral filter kernel as shown below. In this way, there is no need to read the input of bilateral filter (*float depth* variable) from global memory.

```
#pragma OPENCL_EXTENSION cl_altera_channels : enable

//channel float dpth size of 320*240;
channel float dpth  __attribute__((depth(76800)));

__attribute__((task))
__kernel void AOCmm2metersKernel(
    __global float * restrict depth,
    const uint depthSize_x ,
    const uint depthSize_y ,
    const __global ushort * restrict in ,
    const uint inSize_x ,
    const int ratio )
{
  for(uint pixel_y=0;pixel_y<depthSize_y;pixel_y++){
    #pragma unroll 5
    for(uint pixel_x=0;pixel_x<depthSize_x;pixel_x++){
      depth[pixel_x + depthSize_x * pixel_y] = in[pixel_x * ratio + inSize_x * pixel_y * ratio] / 1000.0f;

    }
  }
   for(uint j=0;j<320*240;j++){    //writing mm2meters output to the channel
      write_channel_altera(dpth, depth[j]);
  }


}

__attribute__((task))
__kernel void AOCbilateralFilterkernel( __global float * restrict out,
    //const __global float * restrict in,        //no global input needed
    const float mult_result,
    const int r ,
    const uint2 outSize) {


  float in[320*240];
  for(uint j=0;j<320*240;j++){

    in[j]=read_channel_altera(dpth);      //reading the mm2meters output from the channel

  }

```

Figure 5.3:mm2meters-bilateral filter code channel extended

Analysis:
Benefits of using channels is evident. Not only data transfer efficiency has improved, the input to Bilateral Filter kernel (*float depth*) is now stored in private memory, which can be accessed with the smallest latency. This compromises the addition time added by introducing the for-loop to move the data to from the channel to the kernel.
The total processing time of group1 has decreased from 1.938 milliseconds to 1.186 millisecond, resulting in 1.63x speedup. The main benefit however, is in the resource usage of the kernels.

Area usage	without channel	with channel
Logic utilization	113%	75%
Dedicated logic registers	52%	34%
Memory blocks	108%	81%
DSP blocks	32%	32%

Table 5.4: % of resource usage of optimized mm2meters and bilateral filter with and without channels (kernel resources above 100% is only possible when compiled with --high-effort flag)

Introducing channels have reduced the logic utilization by 38%, memory blocks by 27% and dedicated logic registers by 18%. The ATE has not changed and remained at 0.0202 meters.

2. Group 2 (*halfsample*, *depth2vertex*, *vertex2normal*, *track*): Implementation of channels in this set of kernels is not possible due to the nature of the code:
    a. Half sample kernel gets called 3 times in each frame (default SLAMBench) and the output of each call is fed back as input to the kernel. Channels within a kernel can **only** be in read or write mode only and therefore, it is not possible to implements channel for this kernel.
    b. *Depth2vertex* to *Vertex2Normal*: To implement channels for this set, a variable length channel is needed, which is not supported by Intel FPGA SDK. An alternative way could be using "case statements" to feed the data to the relevant channel, however doing this removes the kernels from fully-pipelined state.
3. Group 3 (*reduce*, *integrate*): These two kernels do not get called the same number of times and therefore, implementation of channel is not possible (Filled channel must be emptied before it can be used again).

5.3 Combining kernels

While the benefit of using channel extension is evident, a more aggressive optimization is combining multiple kernels into one if possible, as Altera Guide suggests. Doing this reduces the number of transfers and more importantly, both kernels get executed in more paralleled fashion.
Before analysing what kernels can be combined, the criteria's for being able to combine kernels are: 1) For NDRange kernels, the iterations space of the kernels must completely match 2) the number of times the kernels get called (in a frame) are the same.

This is only possible for the following kernels:

1. *Mm2meters* and *Bilateral Filter*: The optimized two kernels can be combined without any issues. This code is shown in the Appendix. The total processing time for this kernel has decreased from 1.938 ms to 1.485 ms (1.3x speedup). There are also 26% and 39% reduction in logic utilization and memory block resource usage as well. ATE has not changed and remained at 0.0206 meters.

Area usage	Not combined	Combined
Logic utilization	113%	87%
Dedicated logic registers	52%	43%
Memory blocks	108%	69%
DSP blocks	32%	32%

Table 5.5: % of resource usage of optimized mm2meters and bilateral filter combined and not combined (kernel resources above 100% is only possible when compiled with --high-effort flag)

2. *Depth2vertex* and *Vertex2Normal*: Implementation of channels was not possible for these two kernels; however, they can safely be combined to a single kernel. The combined kernel can be found in Appendix. Combining the kernels decreased the total processing time (transfer time + execution time) from 1.04 milliseconds to 0.824 milliseconds. There is also 8% decrease in logic utilization, while the rest of the resources usage stay the same. ATE has not changed and remained at 0.0205 meters.

Area usage	Not combined	Combined
Logic utilization	68%	60%
Dedicated logic registers	28%	28%
Memory blocks	34%	35%
DSP blocks	6%	6%

Table 5.6: % of resource usage of optimized depth2vertex and vertex2normal combined and not combined

Concurrent kernels

By assigning different command queues, it is possible to execute different kernels simultaneously in a multi-threaded host application. However, this optimization is not possible in KinectFusion algorithm since all I/O of all kernels are dependent to each other (the next kernel can only start when the previous kernel is completed). Furthermore, processing frames in parallel is also not possible because the application executes in process-every-frame mode (fetches the next frame once the previous frame is finished).

5.4 Best match

To come up with the best overall KinectFusion design with the targeted FPGA, there are two main questions that needs to be answered.

1. What kernels are worth sending to the FPGA device for computation?
2. Is it possible to accelerate the kernels that are running on the host CPU?

To answer the following questions, it is fundamental to understand the maximum performance benefit we can achieve from KinectFusion OpenCL application. **Amdahl's law** specifies that the overall maximum theoretical speed up (S) one can expect by parallelizing a portion of a sequential code is:

$$S = \frac{1}{(1-P) + P/N}$$

Where P is the proportion of total sequential execution time taken by the part of the code that can be parallelized and N is the number of cores the parallel part of the code runs on. In the context of KinectFusion algorithm, kernels that cannot be parallelized enough on the FPGA are better off executed on the host CPU. Thus, the focus should be on finding ways of parallelizing code on the host CPU first and send the data to the FPGA only if further acceleration can be obtained.

The amount of speedup that can be achieved on C++ sequential code, depends on how many cores are available on the host machine. Techniques likes parallel for-loops (available at Intel Threading Building Blocks *TBB::parallel_for* library) can be used to break the iteration space in to chucks and run them in parallel. Task groups (available in *TBB:task_group*) is also another method to specify a set of heterogenous C++ tasks that can run in parallel. A simpler technique that harness the power of multi-core CPUs is the compiler extension OpenMP, that can be used to add parallelism to serial C++ program, without much change added to the original code.

To accelerate the host C++ KinectFusion kernels, they have all been replaced with their OpenMP implementation. Table below shows the OpenMP execution time of all kernels on the FPGA server (*snowball2 with 4 cores*) machine and their comparison with total FPGA optimized timings. By total here it is meant the write to FPGA, FPGA execution and read back from FPGA time.

kernel name	4 core OpenMP (snowball2)	FPGA optimized total per-kernel time (snowball2)
mm2	0.245	0.874
Bilateral Filter	16.793	1.42
Half sample	0.13	0.4595
Depth2vertex	0.263	0.5102
Vertex2Normal	0.43	0.857
Track	0.568	1.416
Reduce	0.275	1.363
Integrate	105.558	54.731
Raycast	63.906	164.305
Render depth	0.274	0.631
Render Track	0.42	1.597
Render Volume	64.862	153.504

Table 5.7: Comparison between OpenMP and FPGA total time (ms) on the same machine

What can be seen is that, for many of the kernels it is better to execute them on the 4-core host machine than sending them to the FPGA device for computation. However, for *bilateral filter* and *integrate* kernels, it is certainly worth sending the kernels to the FPGA device. These two optimized kernels also fill the whole logic utilization of the FPGA. Given the resources available and the optimizations made, this model is the best that can be achieved. In this model, all kernels are accelerated using OpenMP apart from *bilateral filter* and *integrate* kernels, which are sent to the FPGA device for acceleration. This model can be executed at an average of 62.63 milliseconds per frame (16 frames per second). The ATE of this model is acceptable at 0.0207 meters.

Logic utilization	**107%**
Dedicated logic registers	**50%**
Memory blocks	**65%**
DSP blocks	**21%**
Frame rate	**8.26 FPS**
ATE	**0.0207 meters**

Table 5.8: FPGA best model parameters

Finally, if the kernels were executed on the boxer4 machine (16 cores), then the pattern above changes. Doing the above comparison for boxer4-OpenMP and FPGA-optimized timing, the best model is executing *bilateral filter*, *half sample* and *depth2vertex* kernels on the FPGA, and the rest on OpenMP host CPUs. Just to clarify, the consecutive nature of the chosen kernels makes FPGA kernels faster than OpenMP host, due to the elimination of transfer times. Frame rates achieved for these models above and how they compare with all other platforms is described in the Final evaluation section.

6 Final evaluation

SLAMBench provides a table, where, just like this project, it has evaluated the performance of KinectFusion algorithm running on 5 different platforms with default parameters and ICL-NUIM dataset. Details of these platforms are as follows:

Machine names	TITAN	GTX870M	TK1	ODROID (XU3)	Arndale
Machine type	Desktop	Laptop	Embedded	Embedded	Embedded
CPU	i7 Haswell	i7 Haswell	NVIDIA 4-Plus-1	Exynos 5422	Exynos 5250
CPU cores	4	4	4 (Cortex-A15) + 1	4 (Cortex-A15) + 4 (Cortex-A7)	2 (Cortex-A15)
CPU GHz	3.5	2.4	2.3	1.8	1.7
GPU	NVIDIA TITAN	NVIDIA GTX 870M	NVIDIA Tegra K1	ARM Mali-T628-MP6	ARM Mali-T604-MP4
GPU architecture	Kepler	Kepler	Kepler	Midgard 2nd gen.	Midgard 1st gen.
GPU FPU32s	2688	1344	192	60	40
GPU MHz	837	941	852	600	533
GPU GFLOPS (SP)	4500	2520	330	60+30 (72+36)	60 (71)
Language	CUDA/OpenCL/C++	CUDA/OpenCL/C++	CUDA/C++	OpenCL/C++	OpenCL/C++
OpenCL version	1.1	1.1	n/a	1.1	1.1
Toolkit version	CUDA 5.5	CUDA 5.5	CUDA 6.0	Mali SDK1.1.	Mali SDK1.1
Ubuntu OS (kernel)	13.04 (3.8.0)	14.04 (3.13.0)	14.04 (3.10.24)	14.04 (3.10.53)	12.04 (3.11.0)

Table 6.0: SLAMBench published paper platforms

Execution time:

The main focus of this project has been on execution time. The execution time of the kernels evaluate the frame rate and how fast we can run SLAM on the platforms. Performance evaluated (Average execution time in seconds) of the evaluated platforms in this project can be summarized in the table 6.1 below (Frame rates on top of each bar).

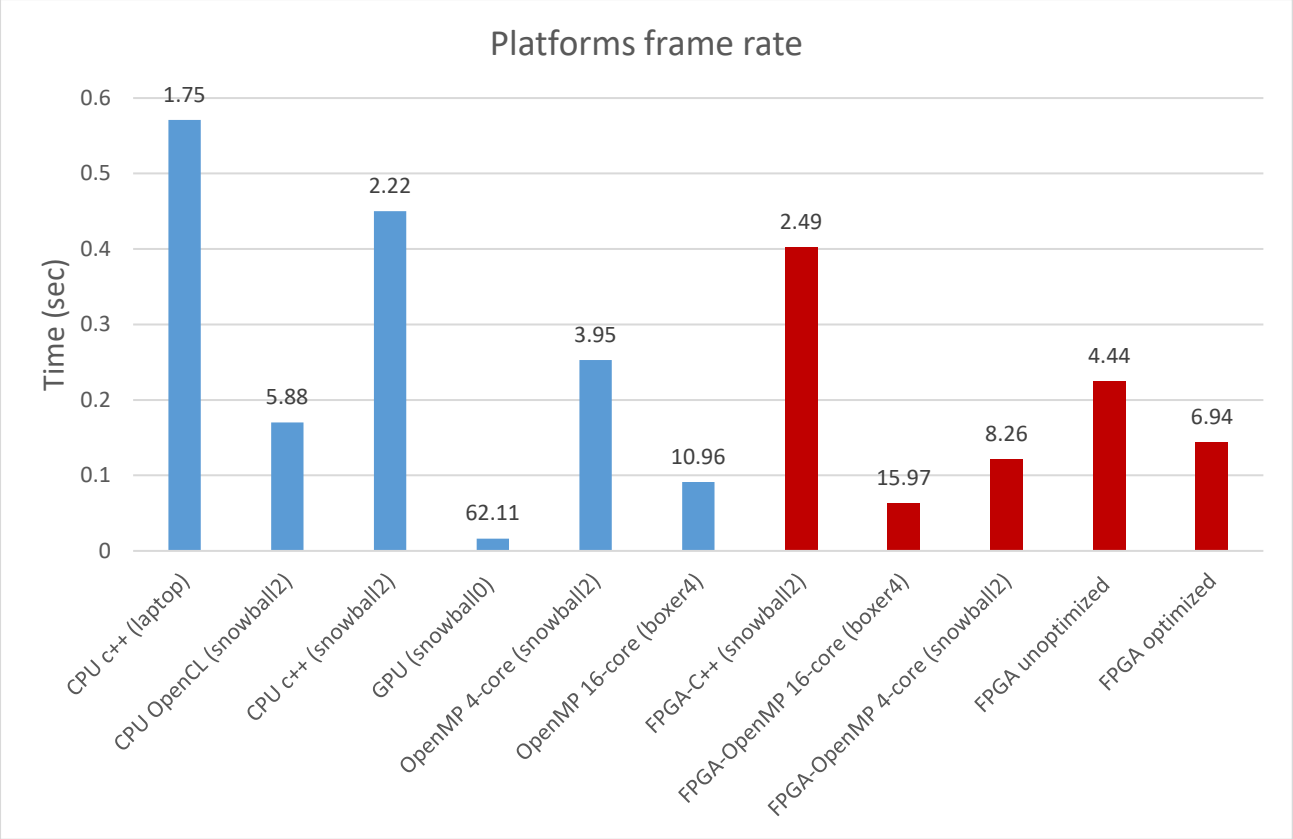


Figure 6.1: Frame rate of all platforms and combinations evaluated in this project

OpenMP snowball2 and *OpenMP boxer4* are measured time when whole application runs in OpenMP on those platforms. *FPGA-C++* platform is the average timing of all kernels, when only one kernel is executed on the FPGA device, while the rest sequentially runs on host CPU (just like per-kernel evaluation). *FPGA-OpenMP* (4-core-snowball2) is the best model with FPGAs, that has been synthesized and tested in this project, achieving a frame rate of 8.26 FPS. *FPGA-OpenMP* (16-core-boxer4) is the predicted performance if FPGA device was available on boxer4 server, which is at 15.97 FPS. *FPGA unoptimized* and *FPGA optimized* platforms are predicted estimated performance if all kernels could fit on the targeted FPGA device.

The same pattern that was seen for single kernels evaluation can be seen here as well. Among the non-FPGA platforms (blue bars) Sequential C++ has the worst performance amongst all design with 1.75 and 2.22 frame rates on laptop and snowball2 machines respectively. This is followed by FPGA-C++ platform (2.49 FPS), where there is slight increase of accelerating a single kernel on the FPGA device. The GPU has the highest frame rate among all the platforms (62 FPS).

What is important to point out here, is the predicted unoptimized FPGA performance (4.44 FPS) is worse than the multi-core CPU (CPU OpenCL with 5.88 FPS) platform. Optimizations applied to the design has made predicted FPGA optimized frame rate (6.94 FPS) faster than multi-core CPU. Another interesting fact, is the significant improvement made, once FPGA is used to accelerate suitable kernels. Once appropriate kernels are mapped to hardware, the OpenMP design on snowball2 frame rate rises from 3.95 FPS to 8.26 FPS. (10.96 to 15.97 on boxer4).

Comparing our frame rates with SLAMBench results:

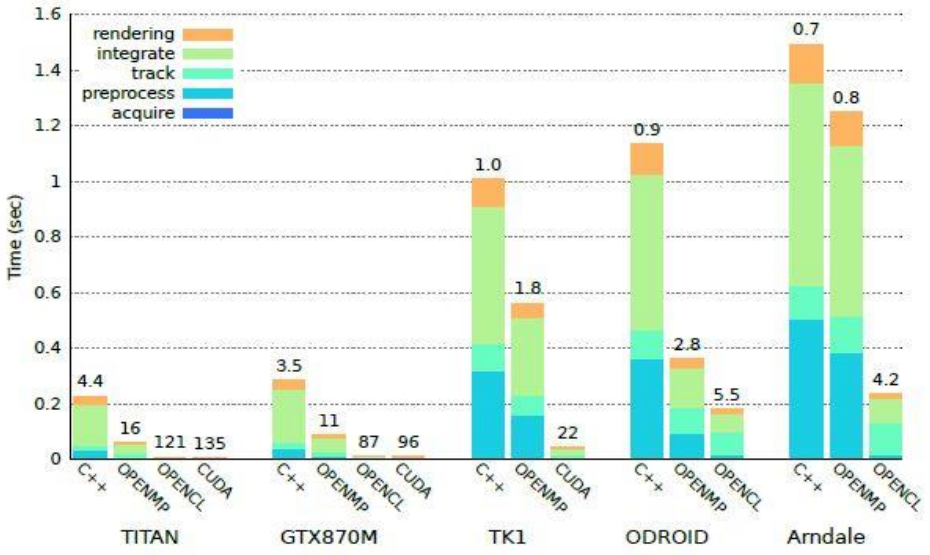


Fig. 4: High-level SLAM building block elapsed times. FPS is reported on top of each histogram.

Figure 6.2: SLAMBench results on different platforms and implementations [4]

FPGA unoptimized frame rate (4.44 FPS) again here is slower than OpenCL frame rate of all platforms. However, the optimizations applied made the FPGA optimized frame rate (6.94 FPS) faster than OpenCL Arndale (4.2 FPS) and Odroid (5.5 FPS). The predicted combined FPGA-OpenMP on 16-core machine, achieves a higher frame rate than all C++, OpenMP platforms including TITAN and GTX870M.

It can be concluded that if enough silicon is available on FPGAs, they can perform a reasonable frame rate of 7 frames per second. Moreover, the targeted FPGA is suitable for single-kernel accelerator of SLAM algorithms, and can achieve up to 15 FPS if they are combined with multi-core CPUs. As mentioned several times, the most time-consuming kernels on the FPGA (*raycast*, *render volume*) execute faster on GPUs as they depend on *position* and *view* that are not known at run-time. When executed on GPUs, each point in these kernels rendered independently while hiding the memory latencies. In general, Slow Off-chip memory access of FPGAs makes them not as fast as GPUs in SLAM applications.

ATE:

Accuracy of SLAM application is an important factor in overall design and has been evaluated constantly throughout the project. The comparison between ATE of FPGA optimized and unoptimized has been discussed before. Comparison between different platforms ATEs and average FPGA kernels ATE is shown below:

platform	ATE (meters)
CPU C++ (laptop)	0.0205
CPU OpenCL (snowball2)	0.0206
CPU C++ (snowball2)	0.0205
GPU (snowball0)	0.0205
OpenMP 4-core (snowball2)	0.0205
OpenMP 16-core (boxer4)	0.0205
FPGA-C++ (snowball2)	0.0205
FPGA-OpenMP 4-core (snowball2)	0.0205

Table 6.3: ATE of all platforms (meters) in this project

As can be seen, the ATE of all platforms in the evaluation are acceptable and around 2 centimetres for all platforms, achieving tracking of the camera from frame 5, without losing the camera tracking throughout the whole 882 frames. ATE of unoptimized and optimized FPGA cannot be evaluated since the performance is an estimation if kernels could fit the device. Comparing that with SLAMBench platforms:

TITAN				GTX870M				TK1			ODROID			Arndale		
C++	OMP	OCL	CUDA	C++	OMP	OCL	CUDA	C++	OMP	CUDA	C++	OMP	OCL	C++	OMP	OCL
2.07	2.07	2.07	2.07	2.07	2.07	2.07	2.07	2.06	2.06	2.07	2.06	2.06	2.01	2.06	2.06	2.07

Table 6.4: ATE of SLAMBench platforms

ATE of SLAMBench platforms are around 2.07 centimetres. The ATE parameter has not changed much from platform to platform, and all values are acceptable.

Energy:

Another important factor in SLAM applications is energy consumption. Figure below shows the documented energy consumption of evaluated platforms based on their data sheet.

Platform	Power (Watt)
Stratix V D5 FPGA	<25
AMD FirePro™ W5000 Graphics	75
Intel(R) Core(TM) i3-2130 3.40GHz Quadcore CPU	65

Figure 6.5: Datasheet power consumption (Watt) of the platforms

It is clear that FPGAs have clear advantage here, consuming a third of the power of GPUs. This is essential for robot SLAM applications, where power is a crucial factor. Exact Power consumption of KinectFusion algorithm can be obtained by plugging a current meter and the above are just datasheet values. Intel FPGA SDK does not generate a power estimation of the hardware generated directly (in the .aocx directory).

Area:

This factor is only applicable to FPGA devices. Compared to the FPGA market today (June 2017), D5 FPGA is an old FPGA (DDR3 2 x 4GB SDRAM). This is a real issue when it comes to floating point arithmetic intensive algorithms like KinectFusion as synthesizing floats is very expensive on FPGAs. The GPU has a clear advantage there. Arria 10 devices have hard floating point DSPs and will compare much better for KinectFusion SLAM algorithm.

Cost:

Below values are based on (June 2017) market prices of the platforms.

Platform	Cost ($)
Stratix V D5 FPGA	$4995
AMD FirePro™ W5000 Graphics	$339
Intel(R) Core(TM) i3-2130 3.40GHz QuadCore CPU	$276

Table 6.6: Cost of buying the platforms as of June 2017

Considering the cost of all hardware platforms, the targeted FPGA is much more expensive than the GPU and Quad-Core CPU. This could be an factor in a SLAM application and it is worth taking in to account when choosing the required hardware platform.

7 Conclusion

Performance of each kernel of KinectFusion algorithm has first been evaluated on GPU, CPUs and FPGA platform. The project has then applied several optimization strategies to per-kernels and analysed how performance metrics have changed. The kernels are then grouped together and again optimization strategies to grouped-kernels have been applied and evaluated. The best accelerated model of FPGA-CPU platform has been introduced and finally, all hardware-accelerated platforms have been evaluated with regards to frame rate, accuracy (ATE), power consumption and cost of running the KinectFusion SLAM algorithm.

The pattern that could be seen in per-kernel evaluation of KinectFusion is that serial CPUs achieve the lowest frame rate running the KinectFusion algorithm at 1.7 FPS. With FPGA-specific optimizations applied, it is presented that the per-kernel frame rates can outperform the quadcore CPUs in kernel execution time. The slow off chip memory access of FPGAs, and the dependencies of the algorithm on parameters that are known at FPGA compile time, makes GPU a better solution to the FPGA with regards to execution time. Most of the memory accesses within the KinectFusion kernels access the memory in random access patterns, which makes GPUs memory architecture more suitable for these types of algorithms in terms of execution time.

The resource limit of the targeted FPGA prevents synthesizing the whole design on the FPGA device. The FPGA however, can be used as a suitable platform for accelerating parts (kernel) of the SLAM algorithm. Combined with CPUs, the best model has achieved 4x speedup compared to software version running on a single CPU. The optimal system presented is capable of running KinectFusion algorithm at 8 frame per second at a resolution of 320x240. The proves that FPGAs are capable of brining complex algorithms like SLAM in to embedded space, with less power consumption than GPUs and more performance that other multi-core CPU solutions.

.7.1 Future work

The work presented in this project has successfully evaluated the hardware accelerated SLAM algorithm in SLAMBench benchmark. The project has mapped the algorithm to the FPGA device and applied FPGA-specific optimizations to the algorithm, crafting new version of KinectFusion algorithm for FPGAs. As explained before, large amount of time was spent writing host-device code outside SLAMBench framework (by writing and reading from text files) for issues faced with tuning SLAMBench to compile for FPGAs. This work, paves the

way for future work on FPGAs in SLAM applications and particularly in SLAMBench..All implemented code can be accessed on the GitHub at [32] Potential future work in this project are as follows:

- Scope of optimization for *Raycast*, *RenderVolume* and *integrate* kernels, which occupy 99% of FPGA execution time, was low. Although these kernels map better to GPUs , they can be rewritten from scratch, while having FPGA specifics in mind. Some parts of the kernels can be approximated to optimize their performance.
- As mentioned before, the targeted FPGA suffers from resource limits to run the whole algorithm. An FPGA with more memory could be evaluated for future projects.
- One optimization technique could be splitting the kernel's processing between the host and FPGA in parallel. This means that the FPGA device can process half of the frame while the other half is being processed on the host CPU.
- The hardware resources limit of the FPGA is mainly because KinectFusion contains heavy floating point arithmetic and FPGAs contain a lot of logic for implementing floating point operations. To increase the hardware resources, fixed point representations of the data could be implemented instead. It is important to mention that OpenCL does not support fixed point representation, and therefore this is only possible by using integer data types. To do this, the data resolution required to do the KinectFusion calculations could be obtained. Since only 8,16,32 and 64-bit fixed data types are available in OpenCL, appropriate masking techniques should be applied to save resources [30].
- Arria 10 FPGA devices are more suitable for SLAM applications because of hard floating point DSP blocks support., enabling 1.5 TFLOPs (Tera Floating-point Operations Per Second)
- Majority of the work done in this project was done with the old version of Intel OpenCL (version 15). The newer version of Intel FPGA SDK (version 17) provides much better optimization reports and suggestions for the kernels. The newer version provides HTML reporting and allows profiling the kernels. The optimized kernels of this project can be profiled by instrumenting performance counters to gather performance information, where it can be viewed by a profiler GUI. This could identify performance bottlenecks of the optimized kernels.
- The optimization path followed in this project was focused on execution time-resources comparison. A power oriented design could be evaluated and added to SLAMBench for FPGAs.

References

[1] J. a. L. J. J. Folkesson, “Autonomy through {SLAM} for an Underwater Robot,” in *Robotics Research,* 2011.

[2] G. S. G. P. P. R. C. F. Auat Cheein, “Optimized EIF-SLAM algorithm for precision agriculture mapping based on stems detection,” *Computers and Electronics in Agriculture,* vol. 78, no. 2, pp. Pages 195-207, September 2011.

[3] A. Davison, “15 Years of Visual SLAM,” Robot Vision Group and Dyson Robotics Laboratory, 18 December 2015. [Online]. Available: http://wp.doc.ic.ac.uk/thefutureofslam/wp-content/uploads/sites/93/2015/12/slides_ajd.pdf. [Accessed 04 June 2017].

[4] B. B. M. Z. Z. J. M. A. N. P. H. J. K. A. J. D. M. L. M. F. P. O. G. R. N. T. S. F. Luigi Nardi, “Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM,” *2015 IEEE International Conference on Robotics and Automation (ICRA),* pp. 5738-5790, May 2015.

[5] L. D'Alfonso, A. Griffo,P. Muraca,P. Pugliese, “A SLAM algorithm for indoor mobile robot localization using an Extended Kalman filter and a segment based environment mapping,” *2013 16th International Conference on Advanced Robotics (ICAR),* pp. 1-6, Nov 2013.

[6] N. Y. Mehmet Korkmaz 1, “Comparison of the SLAM algorithms: Hangar experiments,” no. MATEC Web of Conferences, 2016.

[7] P. E. M. G. a. J. J. L. Robert Sim, “Vision-based SLAM using the Rao-Blackwellised Particle Filter,” University of British Columbia, Vancouver,, 2016.

[8] C. S. a. W. B. Giorgio Grisetti, “Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters,” *IEEE TRANSACTIONS ON ROBOTICS,* vol. 23, February 2007.

[9] K. Boikos and C. S. Bouganis, “Semi-dense SLAM on an FPGA SoC,” *2016 26th International Conference on Field Programmable Logic and Applications (FPL),* pp. 1-4, Aug 2016.

[10] Yusuke Misono,Yoshitaka Goto,Yuki Tarutoko,Kazuyuki Kobayashi,Kajiro Watanabe, “Development of laser rangefinder-based SLAM algorithm for mobile robot navigation,” *SICE Annual Conference 2007,* pp. 392-396, Sept 2007.

[11] Juan D. Tardós,Paul M. Newman, “Robust Mapping and Localization in Indoor Environments Using Sonar Data,” *THE INTERNATIONAL JOURNAL OF ROBOTICS RESEARCH,* vol. 21, no. 4, pp. 311-330, April 2012.

[12] J. W. a. W. Chen, “An Improved Extended Information Filter SLAM Algorithm Based on Omnidirectional Vision,” *Journal of Applied Mathematics,* vol. 2014, no. 948505, p. 10, 2014.

[13] Seongsoo Lee and Sukhan Lee and Jason Jeongsuk Yoon, “Illumination-Invariant Localization Based on Upward Looking Scenes for Low-Cost Indoor Robots,” *Advanced Robotics,* vol. 26, no. 13, pp. 1443-1469, 2012.

[14] J. Wetherall and M. Taylor and D. Hurley-Smith, “Investigation into the effects of transmission-channel fidelity loss in RGBD sensor data for SLAM,” *2015 International Conference on Systems, Signals and Image Processing (IWSSIP),* pp. 81-84, Sept 2015.

[15] D. Honegger and H. Oleynikova and M. Pollefeys, “Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU,” *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, no. 2153-0858, pp. 4930-4935, Sept 2014.

[16] S. Jin and J. Cho and X. D. Pham and K. M. Lee and S. K. Park and M. Kim and J. W. Jeon, “FPGA Design and Implementation of a Real-Time Stereo Vision System,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 1, pp. 15-26, Jan 2010.

[17] D. Bouris and A. Nikitakis and I. Papaefstathiou, “Fast and Efficient FPGA-Based Feature Detection Employing the SURF Algorithm,” *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 3-10, May 2010.

[18] E. T. L. P. Grigorios Mingas, “An FPGA implementation of the SMG-SLAM algorithm,” *Microprocessors and Microsystems*, vol. 36, no. 0141-9331, pp. 190 - 204, 2012.

[19] D. T. T. a. J. P. a. M. Devy, “FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM,” *2014 International Conference on ReConFigurable Computing and FPGAs*, no. 2325-6532, pp. 1-6, Dec 2014.

[20] Khronos Team, “The open standard for parallel programming of heterogeneous systems,” Khonos Group, [Online]. Available: https://www.khronos.org/opencl/. [Accessed 17 Jan 2017].

[21] Khronos Group, “The OpenCL Specification,” 2.2 ed., Khronos OpenCL Working Group, 2016, pp. 25-61.

[22] D. Watt, “A Quick Guide to Writing OpenCL Kernels for PowerVR Rogue GPUs,” Multimedia Strategy Imagination Technologies, 22 Jan 2017. [Online]. Available: https://www.embedded-vision.com/industry-analysis/technical-articles/quick-guide-writing-opencl-kernels-powervr-rogue-gpus. [Accessed 02 June 2017].

[23] A. Inc, “OpenCL Programming Guide for Mac,” 06 June 2017. [Online]. Available: https://developer.apple.com/library/content/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCLLionMemoryObjects/OpenCLLionMemoryObjects.html#//apple_ref/doc/uid/TP40008312-CH103-SW4. [Accessed 06 June 2017].

[24] J. E. S. a. D. G. a. G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” *Computing in Science Engineering*, vol. 2, no. 3, pp. 66-73, May 2010.

[25] deklerkmc, “OpenCL Framework,” 24 July 2013. [Online]. Available: http://mygsoc.blogspot.co.uk/2013/07/opencl-framework.html. [Accessed 19 Jan 2017].

[26] N. N. K. H. K. H. Lee Joo Hwan, “OpenCL Performance Evaluation on Modern Multicore CPUs,” *Scientific Programming*, vol. 2015, no. 859491, p. 20, 2015.

[27] C. Maxfield, “OpenCL S/W dev kit for FPGAs from Altera,” 11 May 2012. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1317510. [Accessed 23 Jan 2017].

[28] H. R. a. M. N. a. S. A. a. M. M. a. M. S. Zohouri, “Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs,” *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, vol. 35, no. 12, pp. 35:1--35:12, 2016.

[29] Intel, “Intel FPGA SDK for OpenCL Programming Guide,” 2016.

[30] Altera, “Altera SDK for OpenCL Best Practices Guide,” 2015.

[31] S. Iannace, “SLAMBench: evaluating OpenCL-based FPGA design for SLAM application,” 2015.

[32] A. Shafiei, “Github,” Github, [Online]. Available: https://github.com/arshansh/slambench. [Accessed 02 June 2017].

[33] “Intel SDK for OpenCL practice examples,” Intel, [Online]. Available: https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html. [Accessed 20 Jan 2017].

Appendix

Vertex2Normal unoptimized

```
666 __kernel void vertex2normalKernel( __global float * normal,    // float3
667     const uint2 normalSize,
668     const __global float * vertex ,
669     const uint2 vertexSize ) {  // float3
670
671   uint2 pixel = (uint2) (get_global_id(0),get_global_id(1));
672
673   if(pixel.x >= vertexSize.x || pixel.y >= vertexSize.y )
674   return;
675
676
677   uint2 vleft = (uint2)(max((int)(pixel.x)-1,0), pixel.y);
678   uint2 vright = (uint2)(min(pixel.x+1,vertexSize.x-1), pixel.y);
679   uint2 vup = (uint2)(pixel.x, max((int)(pixel.y)-1,0));
680   uint2 vdown = (uint2)(pixel.x, min(pixel.y+1,vertexSize.y-1));
681
682   const float3 left = vload3(vleft.x + vertexSize.x * vleft.y,vertex);
683   const float3 right = vload3(vright.x + vertexSize.x * vright.y,vertex);
684   const float3 up = vload3(vup.x + vertexSize.x * vup.y,vertex);
685   const float3 down = vload3(vdown.x + vertexSize.x * vdown.y,vertex);
686
687   if(left.z == 0 || right.z == 0|| up.z ==0 || down.z == 0) {
688
689     vstore3((float3)(INVALID,INVALID,INVALID),pixel.x + normalSize.x * pixel.y,normal);
690     return;
691   }
692   const float3 dxv = right - left;
693   const float3 dyv = down - up;
694   vstore3((float3) normalize(cross(dyv, dxv)), pixel.x + pixel.y * normalSize.x, normal );
695
696 }
```

Vertex2Normal optimized

```
int find_min(int a,int b){

  int min=a;
  if(a>b){
    min=b;
  }
  return min;
}
int find_max(int a,int b){

  int max=a;
  if(b>a){
    max=b;
  }
  return max;
}
#define INVALID -2

__kernel void AOCvertex2normalKernel( __global float * restrict normal,    // float3
    const uint2 normalSize,
    const __global float * restrict vertex ,
    const uint2 vertexSize,
    const uint2 outSize ) {  // float3

  for(int pixel_y=0;pixel_y<outSize.y;pixel_y++){  //single-work-item conversion
    for(int pixel_x=0;pixel_x<outSize.x;pixel_x++){

        if(pixel_x < vertexSize.x && pixel_y < vertexSize.y ){

        uint2 vleft = (uint2)(find_max((int)(pixel_x)-1,0), pixel_y);
        uint2 vright = (uint2)(find_min(pixel_x+1,vertexSize.x-1), pixel_y);
        uint2 vup = (uint2)(pixel_x, find_max((int)(pixel_y)-1,0));
        uint2 vdown = (uint2)(pixel_x, find_min(pixel_y+1,vertexSize.y-1));

        const float3 left = vload3(vleft.x + vertexSize.x * vleft.y,vertex);
        const float3 right = vload3(vright.x + vertexSize.x * vright.y,vertex);
        const float3 up = vload3(vup.x + vertexSize.x * vup.y,vertex);
        const float3 down = vload3(vdown.x + vertexSize.x * vdown.y,vertex);

        if(left.z == 0 || right.z == 0|| up.z ==0 || down.z == 0) {
          vstore3((float3)(INVALID,INVALID,INVALID),pixel_x + normalSize.x * pixel_y,normal);
        }
        else{
          const float3 dxv = right - left;
          const float3 dyv = down - up;
          vstore3((float3) normalize(cross(dyv, dxv)), pixel_x + pixel_y * normalSize.x, normal );
        }
       }
    }
  }
}
```

Reduce unoptimized

```
typedef struct sTrackData {
    int result;
    float error;
    float J[6];
} TrackData;


__kernel void AOCreduceKernel (
        __global float * out,
          global const TrackData * J,
        const uint2 JSize,
        const uint2 size,
          local float * S
) {

    uint blockIdx = get_group_id(0);
    uint blockDim = get_local_size(0);
    uint threadIdx = get_local_id(0);
    uint gridDim = get_num_groups(0);

    const uint sline = threadIdx;

    float sums[32];
    float * jtj = sums + 7;
    float * info = sums + 28;

    for(uint i = 0; i < 32; ++i)
    sums[i] = 0.0f;

    for(uint y = blockIdx; y < size.y; y += gridDim) {
        for(uint x = sline; x < size.x; x += blockDim ) {
            const TrackData row = J[x + y * JSize.x];
            if(row.result < 1) {
                info[1] += row.result == -4 ? 1 : 0;
                info[2] += row.result == -5 ? 1 : 0;
                info[3] += row.result > -4 ? 1 : 0;
                continue;
            }

            // Error part
            sums[0] += row.error * row.error;

            // JTe part
            for(int i = 0; i < 6; ++i)
            sums[i+1] += row.error * row.J[i];

            jtj[0] += row.J[0] * row.J[0];
            jtj[1] += row.J[0] * row.J[1];
            jtj[2] += row.J[0] * row.J[2];
            jtj[3] += row.J[0] * row.J[3];
            jtj[4] += row.J[0] * row.J[4];
            jtj[5] += row.J[0] * row.J[5];

            jtj[6] += row.J[1] * row.J[1];
            jtj[7] += row.J[1] * row.J[2];
            jtj[8] += row.J[1] * row.J[3];
            jtj[9] += row.J[1] * row.J[4];
            jtj[10] += row.J[1] * row.J[5];

            jtj[11] += row.J[2] * row.J[2];
            jtj[12] += row.J[2] * row.J[3];
            jtj[13] += row.J[2] * row.J[4];
            jtj[14] += row.J[2] * row.J[5];

            jtj[15] += row.J[3] * row.J[3];
            jtj[16] += row.J[3] * row.J[4];
            jtj[17] += row.J[3] * row.J[5];

            jtj[18] += row.J[4] * row.J[4];
            jtj[19] += row.J[4] * row.J[5];
```

```
            jtj[20] += row.J[5] * row.J[5];
            // extra info here
            info[0] += 1;
        }
    }

    for(int i = 0; i < 32; ++i) // copy over to shared memory
    S[sline * 32 + i] = sums[i];

    barrier(CLK_LOCAL_MEM_FENCE);

    if(sline < 32) { // sum up columns and copy to global memory in the final 32 threads
        for(unsigned i = 1; i < blockDim; ++i)
        S[sline] += S[i * 32 + sline];
        out[sline+blockIdx*32] = S[sline];
    }
}
```

Reduce optimized

```
typedef struct sTrackData {
    int result;
    float error;
    float J[6];
} TrackData;


__attribute__((reqd_work_group_size(64,1,1)))
__attribute__(  (num_compute_units(2))  )

__kernel void AOCreduceKernel (
        __global float * restrict out,
          global const TrackData * restrict  J,
        const uint2 JSize,
        const uint2 size,
          local float * S
) {

    uint blockIdx = get_group_id(0);
    uint blockDim = get_local_size(0);
    uint threadIdx = get local id(0);
    uint gridDim = get_num_groups(0);

    const uint sline = threadIdx;

    float sums[32];
    float * jtj = sums + 7;
    float * info = sums + 28;


  #pragma unroll
    for(uint i = 0; i < 32; ++i)
    sums[i] = 0.0f;

    for(uint y = blockIdx; y < size.y; y += gridDim) {
        for(uint x = sline; x < size.x; x += blockDim ) {
            const TrackData row = J[x + y * JSize.x];
            if(row.result < 1) {
                info[1] += row.result == -4 ? 1 : 0;
                info[2] += row.result == -5 ? 1 : 0;
                info[3] += row.result > -4 ? 1 : 0;
                continue;
            }

            // Error part
            sums[0] += row.error * row.error;

            // JTe part
      #pragma unroll
            for(int i = 0; i < 6; ++i)
            sums[i+1] += row.error * row.J[i];

            jtj[0] += row.J[0] * row.J[0];
            jtj[1] += row.J[0] * row.J[1];
            jtj[2] += row.J[0] * row.J[2];
            jtj[3] += row.J[0] * row.J[3];
            jtj[4] += row.J[0] * row.J[4];
            jtj[5] += row.J[0] * row.J[5];

            jtj[6] += row.J[1] * row.J[1];
            jtj[7] += row.J[1] * row.J[2];
            jtj[8] += row.J[1] * row.J[3];
            jtj[9] += row.J[1] * row.J[4];
            jtj[10] += row.J[1] * row.J[5];

            jtj[11] += row.J[2] * row.J[2];
            jtj[12] += row.J[2] * row.J[3];
            jtj[13] += row.J[2] * row.J[4];
```

```
            jtj[15] += row.J[3] * row.J[3];
            jtj[16] += row.J[3] * row.J[4];
            jtj[17] += row.J[3] * row.J[5];

            jtj[18] += row.J[4] * row.J[4];
            jtj[19] += row.J[4] * row.J[5];

            jtj[20] += row.J[5] * row.J[5];
            // extra info here
            info[0] += 1;
        }
    }
  #pragma unroll
    for(int i = 0; i < 32; ++i) // copy over to shared memory
    S[sline * 32 + i] = sums[i];

    barrier(CLK_LOCAL_MEM_FENCE);

    if(sline < 32) { // sum up columns and copy to global memory in the final 32 threads
        for(unsigned i = 1; i < blockDim; ++i)
        S[sline] += S[i * 32 + sline];
        out[sline+blockIdx*32] = S[sline];
    }
}
```

Integrate unoptimized

```
448 __kernel void integrateKernel (
449     __global short2 * v_data,
450     const uint3 v_size,
451     const float3 v_dim,
452     __global const float * depth,
453     const uint2 depthSize,
454     const Matrix4 invTrack,
455     const Matrix4 K,
456     const float mu,
457     const float maxweight ,
458     const float3 delta ,
459     const float3 cameraDelta
460 ) {
461
462   Volume vol; vol.data = v_data; vol.size = v_size; vol.dim = v_dim;
463
464   uint3 pix = (uint3) (get_global_id(0),get_global_id(1),0);
465   const int sizex = get_global_size(0);
466
467   float3 pos = Mat4TimeFloat3 (invTrack , posVolume(vol,pix));
468   float3 cameraX = Mat4TimeFloat3 ( K , pos);
469
470   for(pix.z = 0; pix.z < vol.size.z; ++pix.z, pos += delta, cameraX += cameraDelta) {
471     if(pos.z < 0.0001f) // some near plane constraint
472     continue;
473     const float2 pixel = (float2) (cameraX.x/cameraX.z + 0.5f, cameraX.y/cameraX.z + 0.5f);
474
475     if(pixel.x < 0 || pixel.x > depthSize.x-1 || pixel.y < 0 || pixel.y > depthSize.y-1)
476     continue;
477     const uint2 px = (uint2)(pixel.x, pixel.y);
478     float depthpx = depth[px.x + depthSize.x * px.y];
479
480     if(depthpx == 0) continue;
481     const float diff = ((depthpx) - cameraX.z) * sqrt(1+sq(pos.x/pos.z) + sq(pos.y/pos.z));
482
483     if(diff > -mu) {
484       const float sdf = fmin(1.f, diff/mu);
485       float2 data = getVolume(vol,pix);
486       data.x = clamp((data.y*data.x + sdf)/(data.y + 1), -1.f, 1.f);
487       data.y = fmin(data.y+1, maxweight);
488       setVolume(vol,pix, data);
489     }
490   }
491 }
```

Integrate optimized

```
__attribute__((reqd_work_group_size(32,32,1)))  //defining a work group size
__attribute__(  (num_compute_units(4))  )       //replication


__kernel void AOCintegrateKernel (
    __global short2 * restrict v_data,
    const uint3 v_size,
    const float3 v_dim,
    __global const float * restrict depth,
    const uint2 depthSize,
      __global const  Matrix4* restrict invTrack,
      __global const  Matrix4* restrict 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);

      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;
        const float diff = ((depthpx) - cameraX.z) * sqrt(1+sq(pos.x/pos.z) + sq(pos.y/pos.z));

        if(diff > -mu) {
          const float sdf = fmin(1.f, diff/mu);
          float2 data = getVolume(vol,pix);
          data.x = clamp((data.y*data.x + sdf)/(data.y + 1), -1.f, 1.f);
          data.y = fmin(data.y+1, maxweight);
          setVolume(vol,pix, data);
        }
      }
}
```

Raycast

```
typedef struct sMatrix4 {
    float4 data[4];
} Matrix4;

typedef struct sVolume {
    uint3 size;
    float3 dim;
    __global short2 * data;
} Volume;



#define INVALID -2


inline float vs(const uint3 pos, const Volume v) {
    return v.data[pos.x + pos.y * v.size.x + pos.z * v.size.x * v.size.y].x;
}


inline float interp(const float3 pos, const Volume v) {
    const float3 scaled_pos = (float3)((pos.x * v.size.x / v.dim.x) - 0.5f,
            (pos.y * v.size.y / v.dim.y) - 0.5f,
            (pos.z * v.size.z / v.dim.z) - 0.5f);
    float3 basef = (float3)(0);
    const int3 base = convert_int3(floor(scaled_pos));
    const float3 factor = (float3)(fract(scaled_pos, (float3 *) &basef));
    const int3 lower = max(base, (int3)(0));
    const int3 upper = min(base + (int3)(1), convert int3(v.size) - (int3)(1));
    return (((vs((uint3)(lower.x, lower.y, lower.z), v) * (1 - factor.x)
            + vs((uint3)(upper.x, lower.y, lower.z), v) * factor.x)
            * (1 - factor.y)
            + (vs((uint3)(lower.x, upper.y, lower.z), v) * (1 - factor.x)
                    + vs((uint3)(upper.x, upper.y, lower.z), v) * factor.x)
                    * factor.y) * (1 - factor.z)
            + ((vs((uint3)(lower.x, lower.y, upper.z), v) * (1 - factor.x)
                    + vs((uint3)(upper.x, lower.y, upper.z), v) * factor.x)
                    * (1 - factor.y)
                    + (vs((uint3)(lower.x, upper.y, upper.z), v)
                            * (1 - factor.x)
                            + vs((uint3)(upper.x, upper.y, upper.z), v)
                                    * factor.x) * factor.y) * factor.z)
            * 0.00003051944088f;
}


inline float3 grad(float3 pos, const Volume v) {
    const float3 scaled pos = (float3)((pos.x * v.size.x / v.dim.x) - 0.5f,
            (pos.y * v.size.y / v.dim.y) - 0.5f,
            (pos.z * v.size.z / v.dim.z) - 0.5f);
    const int3 base = (int3)(floor(scaled pos.x), floor(scaled pos.y),
            floor(scaled_pos.z));
    const float3 basef = (float3)(0);
    const float3 factor = (float3) fract(scaled pos, (float3 *) &basef);
    const int3 lower_lower = max(base - (int3)(1), (int3)(0));
    const int3 lower_upper = max(base, (int3)(0));
    const int3 upper_lower = min(base + (int3)(1),
            convert_int3(v.size) - (int3)(1));
    const int3 upper upper = min(base + (int3)(2),
            convert_int3(v.size) - (int3)(1));
    const int3 lower = lower_upper;
    const int3 upper = upper lower;

    float3 gradient;

    gradient.x = (((vs((uint3)(upper_lower.x, lower.y, lower.z), v)
            - vs((uint3)(lower lower.x, lower.y, lower.z), v)) * (1 - factor.x)
            + (vs((uint3)(upper_upper.x, lower.y, lower.z), v)
                    - vs((uint3)(lower_upper.x, lower.y, lower.z), v))
                    * factor.x) * (1 - factor.y)
```

```
        + ((vs((uint3)(upper_lower.x, upper.y, lower.z), v)
                - vs((uint3)(lower_lower.x, upper.y, lower.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper_upper.x, upper.y, lower.z), v)
                        - vs((uint3)(lower_upper.x, upper.y, lower.z), v))
                        * factor.x) * factor.y) * (1 - factor.z)
        + (((vs((uint3)(upper_lower.x, lower.y, upper.z), v)
                - vs((uint3)(lower_lower.x, lower.y, upper.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper_upper.x, lower.y, upper.z), v)
                        - vs((uint3)(lower upper.x, lower.y, upper.z), v))
                        * factor.x) * (1 - factor.y)
                + ((vs((uint3)(upper_lower.x, upper.y, upper.z), v)
                        - vs((uint3)(lower lower.x, upper.y, upper.z), v))
                        * (1 - factor.x)
                        + (vs((uint3)(upper_upper.x, upper.y, upper.z), v)
                                - vs(
                                        (uint3)(lower_upper.x, upper.y,
                                                upper.z), v)) * factor.x)
                        * factor.y) * factor.z;

gradient.y = (((vs((uint3)(lower.x, upper lower.y, lower.z), v)
        - vs((uint3)(lower.x, lower_lower.y, lower.z), v)) * (1 - factor.x)
        + (vs((uint3)(upper.x, upper_lower.y, lower.z), v)
                - vs((uint3)(upper.x, lower lower.y, lower.z), v))
                * factor.x) * (1 - factor.y)
        + ((vs((uint3)(lower.x, upper_upper.y, lower.z), v)
                - vs((uint3)(lower.x, lower_upper.y, lower.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, upper upper.y, lower.z), v)
                        - vs((uint3)(upper.x, lower_upper.y, lower.z), v))
                        * factor.x) * factor.y) * (1 - factor.z)
        + (((vs((uint3)(lower.x, upper lower.y, upper.z), v)
                - vs((uint3)(lower.x, lower_lower.y, upper.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, upper lower.y, upper.z), v)
                        - vs((uint3)(upper.x, lower_lower.y, upper.z), v))
                        * factor.x) * (1 - factor.y)
                + ((vs((uint3)(lower.x, upper_upper.y, upper.z), v)
                        - vs((uint3)(lower.x, lower_upper.y, upper.z), v))
                        * (1 - factor.x)
                        + (vs((uint3)(upper.x, upper_upper.y, upper.z), v)
                                - vs(
                                        (uint3)(upper.x, lower upper.y,
                                                upper.z), v)) * factor.x)
                        * factor.y) * factor.z;

gradient.z = (((vs((uint3)(lower.x, lower.y, upper_lower.z), v)
        - vs((uint3)(lower.x, lower.y, lower lower.z), v)) * (1 - factor.x)
        + (vs((uint3)(upper.x, lower.y, upper_lower.z), v)
                - vs((uint3)(upper.x, lower.y, lower_lower.z), v))
                * factor.x) * (1 - factor.y)
        + ((vs((uint3)(lower.x, upper.y, upper_lower.z), v)
                - vs((uint3)(lower.x,upper.y, lower_lower.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, upper.y, upper_lower.z), v)
                        - vs((uint3)(upper.x, upper.y, lower_lower.z), v))
                        * factor.x) * factor.y) * (1 - factor.z)
        + (((vs((uint3)(lower.x, lower.y, upper_upper.z), v)
                - vs((uint3)(lower.x, lower.y, lower upper.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, lower.y, upper_upper.z), v)
                        - vs((uint3)(upper.x, lower.y, lower upper.z), v))
                        * factor.x) * (1 - factor.y)
                + ((vs((uint3)(lower.x, upper.y, upper_upper.z), v)
                        - vs((uint3)(lower.x, upper.y, lower_upper.z), v))
                        * (1 - factor.x)
                        + (vs((uint3)(upper.x, upper.y, upper upper.z), v)
                                - vs(
                                        (uint3)(upper.x, upper.y,
                                                lower_upper.z), v))
```

```
                                    * factor.x) * factor.y) * factor.z;

    return gradient
            * (float3)(v.dim.x / v.size.x, v.dim.y / v.size.y,
                    v.dim.z / v.size.z) * (0.5f * 0.000030519440088f);
}

inline float3 get_translation(  __global const  Matrix4* restrict view) {
    return (float3)(view->data[0].w, view->data[1].w, view->data[2].w);
}


inline float3 myrotate(     __global const  Matrix4* restrict M, const float3 v) {
    return (float3)(dot((float3)(M->data[0].x, M->data[0].y, M->data[0].z), v),
            dot((float3)(M->data[1].x, M->data[1].y, M->data[1].z), v),
            dot((float3)(M->data[2].x, M->data[2].y, M->data[2].z), v));
}

float4 raycast(const Volume v, const uint2 pos, __global  const Matrix4* view,
        const float nearPlane, const float farPlane, const float step,
        const float largestep) {

    const float3 origin = get_translation(view);
    const float3 direction = myrotate(view, (float3)(pos.x, pos.y, 1.f));

    // intersect ray with a box
    //
    // www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm
    // compute intersection of ray with all six bbox planes
    const float3 invR = (float3)(1.0f) / direction;
    const float3 tbot = (float3) - 1 * invR * origin;
    const float3 ttop = invR * (v.dim - origin);

    // re-order intersections to find smallest and largest on each axis
    const float3 tmin = fmin(ttop, tbot);
    const float3 tmax = fmax(ttop, tbot);

    // find the largest tmin and the smallest tmax
    const float largest_tmin = fmax(fmax(tmin.x, tmin.y), fmax(tmin.x, tmin.z));
    const float smallest_tmax = fmin(fmin(tmax.x, tmax.y),
            fmin(tmax.x, tmax.z));

    // check against near and far plane
    const float tnear = fmax(largest_tmin, nearPlane);
    const float tfar = fmin(smallest_tmax, farPlane);

    if (tnear < tfar) {
        // first walk with largesteps until we found a hit
        float t = tnear;
        float stepsize = largestep;
        float f_t = interp(origin + direction * t, v);
        float f_tt = 0;
        if (f_t > 0) { // ups, if we were already in it, then don't render anything here
            for (; t < tfar; t += stepsize) {
                f_tt = interp(origin + direction * t, v);
                if (f_tt < 0)                  // got it, jump out of inner loop
                    break;
                if (f_tt < 0.8f)               // coming closer, reduce stepsize
                    stepsize = step;
                f_t = f_tt;
            }
            if (f_tt < 0) {           // got it, calculate accurate intersection
                t = t + stepsize * f_tt / (f_t - f_tt);
                return (float4)(origin + direction * t, t);
            }
        }
    }

    return (float4)(0);
}
```

```
__kernel void AOCraycastKernel( __global float * pos3D,  //float3
          global float * normal,//float3
         __global short2 * v_data,
         const uint3 v_size,
         const float3 v_dim,
             __global const  Matrix4* view,
         const float nearPlane,
         const float farPlane,
         const float step,
         const float largestep ) {

    const Volume volume = {v_size, v_dim,v_data};

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

    const float4 hit = raycast( volume, pos, view, nearPlane, farPlane, step, largestep );
    const float3 test = as float3(hit);

    if(hit.w > 0.0f ) {
        vstore3(test,pos.x + sizex * pos.y,pos3D);
        float3 surfNorm = grad(test,volume);
        if(length(surfNorm) == 0) {
            //float3 n =  (INVALID,0,0);//vload3(pos.x + sizex * pos.y,normal);
            //n.x=INVALID;
            vstore3((float3)(INVALID,INVALID,INVALID),pos.x + sizex * pos.y,normal);
        } else {
            vstore3(normalize(surfNorm),pos.x + sizex * pos.y,normal);
        }
    } else {
        vstore3((float3)(0),pos.x + sizex * pos.y,pos3D);
        vstore3((float3)(INVALID, INVALID, INVALID),pos.x + sizex * pos.y,normal);
    }
}
```

Render Depth unoptimized

```
__kernel void AOCrenderDepthKernel( __global uchar4 * out,
    __global float * depth,
    const float nearPlane,
    const float farPlane ) {

  const int posx = get_global_id(0);
  const int posy = get_global_id(1);
  const int sizex = get_global_size(0);
  float d= depth[posx + sizex * posy];
  if(d < nearPlane)
  vstore4((uchar4)(255, 255, 255, 0), posx + sizex * posy, (__global uchar*)out); // The forth value in uchar4 is padding for memory alignement and so it is for following
 uchar4
  else {
    if(d > farPlane)
    vstore4((uchar4)(0, 0, 0, 0), posx + sizex * posy, (__global uchar*)out);
    else {
      float h =(d - nearPlane) / (farPlane - nearPlane);
      h *= 6.0f;
      const int sextant = (int)h;
      const float fract = h - sextant;
      const float mid1 = 0.25f + (0.5f*fract);
      const float mid2 = 0.75f - (0.5f*fract);
      switch (sextant)
      {
        case 0: vstore4((uchar4)(191, 255*mid1, 64, 0), posx + sizex * posy, (__global uchar*)out); break;
        case 1: vstore4((uchar4)(255*mid2, 191, 64, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 2: vstore4((uchar4)(64, 191, 255*mid1, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 3: vstore4((uchar4)(64, 255*mid2, 191, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 4: vstore4((uchar4)(255*mid1, 64, 191, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 5: vstore4((uchar4)(191, 64, 255*mid2, 0),posx + sizex * posy ,(__global uchar*)out); break;
      }
    }
  }
}

```

Render Depth optimized

```
__attribute__(  (num_simd_work_items(2))  )
__attribute__((reqd_work_group_size(80,60,1)))
__attribute__(  (num_compute_units(2))  )

__kernel void AOCrenderDepthKernel( __global uchar4 * restrict  out,
    __global float * restrict depth,
    const float nearPlane,
    const float farPlane ) {

  const int posx = get_global_id(0);
  const int posy = get_global_id(1);
  const int sizex = get_global_size(0);


  float d= depth[posx + sizex * posy];
  if(d < nearPlane)
  vstore4((uchar4)(255, 255, 255, 0), posx + sizex * posy, (__global uchar*)out); // The forth value in uchar4 is padding for memory alignement and so it is for following
 uchar4
  else {
    if(d > farPlane)
    vstore4((uchar4)(0, 0, 0, 0), posx + sizex * posy, (__global uchar*)out);
    else {
      float h =(d - nearPlane) / (farPlane - nearPlane);
      h *= 6.0f;
      const int sextant = (int)h;
      const float fract = h - sextant;
      const float mid1 = 0.25f + (0.5f*fract);
      const float mid2 = 0.75f - (0.5f*fract);
      switch (sextant)
      {
        case 0: vstore4((uchar4)(191, 255*mid1, 64, 0), posx + sizex * posy, (__global uchar*)out); break;
        case 1: vstore4((uchar4)(255*mid2, 191, 64, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 2: vstore4((uchar4)(64, 191, 255*mid1, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 3: vstore4((uchar4)(64, 255*mid2, 191, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 4: vstore4((uchar4)(255*mid1, 64, 191, 0),posx + sizex * posy ,(__global uchar*)out); break;
        case 5: vstore4((uchar4)(191, 64, 255*mid2, 0),posx + sizex * posy ,(__global uchar*)out); break;
      }
    }
  }
}
```

Render track unoptimized

```
916 typedef struct sTrackData {
917   int result;
918   float error;
919   float J[6];
920 } TrackData;
921
922
923 __kernel void AOCrenderTrackKernel( __global uchar3 * out,
924     __global const TrackData * data ) {
925
926   const int posx = get_global_id(0);
927   const int posy = get_global_id(1);
928   const int sizex = get_global_size(0);
929
930   switch(data[posx + sizex * posy].result) {
931     // The forth value in uchar4 is padding for memory alignement and so it is for following uchar4
932     case  1: vstore4((uchar4)(128, 128, 128, 0), posx + sizex * posy, (__global uchar*)out); break; // ok  GREY
933     case -1: vstore4((uchar4)(000, 000, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // no input BLACK
934     case -2: vstore4((uchar4)(255, 000, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // not in image RED
935     case -3: vstore4((uchar4)(000, 255, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // no correspondence GREEN
936     case -4: vstore4((uchar4)(000, 000, 255, 0), posx + sizex * posy, (__global uchar*)out); break; // too far away BLUE
937     case -5: vstore4((uchar4)(255, 255, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // wrong normal YELLOW
938     default: vstore4((uchar4)(255, 128, 128, 0), posx + sizex * posy, (__global uchar*)out); return;
939   }
940 }
```

Render Track optimized

```
916 typedef struct sTrackData {
917   int result;
918   float error;
919   float J[6];
920 } TrackData;
921
922
923 __kernel void AOCrenderTrackKernel( __global uchar3 * out,
924     __global const TrackData * data ) {
925
926   const int posx = get_global_id(0);
927   const int posy = get_global_id(1);
928   const int sizex = get_global_size(0);
929
930   switch(data[posx + sizex * posy].result) {
931     // The forth value in uchar4 is padding for memory alignement and so it is for following uchar4
932     case  1: vstore4((uchar4)(128, 128, 128, 0), posx + sizex * posy, (__global uchar*)out); break; // ok  GREY
933     case -1: vstore4((uchar4)(000, 000, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // no input BLACK
934     case -2: vstore4((uchar4)(255, 000, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // not in image RED
935     case -3: vstore4((uchar4)(000, 255, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // no correspondence GREEN
936     case -4: vstore4((uchar4)(000, 000, 255, 0), posx + sizex * posy, (__global uchar*)out); break; // too far away BLUE
937     case -5: vstore4((uchar4)(255, 255, 000, 0), posx + sizex * posy, (__global uchar*)out); break; // wrong normal YELLOW
938     default: vstore4((uchar4)(255, 128, 128, 0), posx + sizex * posy, (__global uchar*)out); return;
939   }
940 }
```

Render Volume unoptimized

```
typedef struct sMatrix4 {
    float4 data[4];
} Matrix4;

typedef struct sVolume {
    uint3 size;
    float3 dim;
    __global short2 * data;
} Volume;



#define INVALID -2


inline float vs(const uint3 pos, const Volume v) {
    return v.data[pos.x + pos.y * v.size.x + pos.z * v.size.x * v.size.y].x;
}


inline float interp(const float3 pos, const Volume v) {
    const float3 scaled_pos = (float3)((pos.x * v.size.x / v.dim.x) - 0.5f,
            (pos.y * v.size.y / v.dim.y) - 0.5f,
            (pos.z * v.size.z / v.dim.z) - 0.5f);
    float3 basef = (float3)(0);
    const int3 base = convert_int3(floor(scaled_pos));
    const float3 factor = (float3)(fract(scaled_pos, (float3 *) &basef));
    const int3 lower = max(base, (int3)(0));
    const int3 upper = min(base + (int3)(1), convert int3(v.size) - (int3)(1));
    return (((vs((uint3)(lower.x, lower.y, lower.z), v) * (1 - factor.x)
            + vs((uint3)(upper.x, lower.y, lower.z), v) * factor.x)
            * (1 - factor.y)
            + (vs((uint3)(lower.x, upper.y, lower.z), v) * (1 - factor.x)
                    + vs((uint3)(upper.x, upper.y, lower.z), v) * factor.x)
                    * factor.y) * (1 - factor.z)
            + ((vs((uint3)(lower.x, lower.y, upper.z), v) * (1 - factor.x)
                    + vs((uint3)(upper.x, lower.y, upper.z), v) * factor.x)
                    * (1 - factor.y)
                    + (vs((uint3)(lower.x, upper.y, upper.z), v)
                            * (1 - factor.x)
                            + vs((uint3)(upper.x, upper.y, upper.z), v)
                                    * factor.x) * factor.y) * factor.z)
            * 0.00003051944088f;
}


inline float3 grad(float3 pos, const Volume v) {
    const float3 scaled pos = (float3)((pos.x * v.size.x / v.dim.x) - 0.5f,
            (pos.y * v.size.y / v.dim.y) - 0.5f,
            (pos.z * v.size.z / v.dim.z) - 0.5f);
    const int3 base = (int3)(floor(scaled pos.x), floor(scaled pos.y),
            floor(scaled_pos.z));
    const float3 basef = (float3)(0);
    const float3 factor = (float3) fract(scaled pos, (float3 *) &basef);
    const int3 lower_lower = max(base - (int3)(1), (int3)(0));
    const int3 lower_upper = max(base, (int3)(0));
    const int3 upper_lower = min(base + (int3)(1),
            convert_int3(v.size) - (int3)(1));
    const int3 upper upper = min(base + (int3)(2),
            convert_int3(v.size) - (int3)(1));
    const int3 lower = lower_upper;
    const int3 upper = upper lower;

    float3 gradient;

    gradient.x = (((vs((uint3)(upper_lower.x, lower.y, lower.z), v)
            - vs((uint3)(lower lower.x, lower.y, lower.z), v)) * (1 - factor.x)
            + (vs((uint3)(upper_upper.x, lower.y, lower.z), v)
                    - vs((uint3)(lower_upper.x, lower.y, lower.z), v))
                    * factor.x) * (1 - factor.y)
```

```
        + ((vs((uint3)(upper_lower.x, upper.y, lower.z), v)
                - vs((uint3)(lower_lower.x, upper.y, lower.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper_upper.x, upper.y, lower.z), v)
                        - vs((uint3)(lower_upper.x, upper.y, lower.z), v))
                        * factor.x) * factor.y) * (1 - factor.z)
        + (((vs((uint3)(upper_lower.x, lower.y, upper.z), v)
                - vs((uint3)(lower_lower.x, lower.y, upper.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper_upper.x, lower.y, upper.z), v)
                        - vs((uint3)(lower upper.x, lower.y, upper.z), v))
                        * factor.x) * (1 - factor.y)
                + ((vs((uint3)(upper_lower.x, upper.y, upper.z), v)
                        - vs((uint3)(lower lower.x, upper.y, upper.z), v))
                        * (1 - factor.x)
                        + (vs((uint3)(upper_upper.x, upper.y, upper.z), v)
                                - vs(
                                        (uint3)(lower_upper.x, upper.y,
                                                upper.z), v)) * factor.x)
                        * factor.y) * factor.z;

gradient.y = (((vs((uint3)(lower.x, upper lower.y, lower.z), v)
        - vs((uint3)(lower.x, lower_lower.y, lower.z), v)) * (1 - factor.x)
        + (vs((uint3)(upper.x, upper_lower.y, lower.z), v)
                - vs((uint3)(upper.x, lower lower.y, lower.z), v))
                * factor.x) * (1 - factor.y)
        + ((vs((uint3)(lower.x, upper_upper.y, lower.z), v)
                - vs((uint3)(lower.x, lower_upper.y, lower.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, upper upper.y, lower.z), v)
                        - vs((uint3)(upper.x, lower_upper.y, lower.z), v))
                        * factor.x) * factor.y) * (1 - factor.z)
        + (((vs((uint3)(lower.x, upper lower.y, upper.z), v)
                - vs((uint3)(lower.x, lower_lower.y, upper.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, upper lower.y, upper.z), v)
                        - vs((uint3)(upper.x, lower_lower.y, upper.z), v))
                        * factor.x) * (1 - factor.y)
                + ((vs((uint3)(lower.x, upper_upper.y, upper.z), v)
                        - vs((uint3)(lower.x, lower_upper.y, upper.z), v))
                        * (1 - factor.x)
                        + (vs((uint3)(upper.x, upper_upper.y, upper.z), v)
                                - vs(
                                        (uint3)(upper.x, lower upper.y,
                                                upper.z), v)) * factor.x)
                        * factor.y) * factor.z;

gradient.z = (((vs((uint3)(lower.x, lower.y, upper_lower.z), v)
        - vs((uint3)(lower.x, lower.y, lower lower.z), v)) * (1 - factor.x)
        + (vs((uint3)(upper.x, lower.y, upper_lower.z), v)
                - vs((uint3)(upper.x, lower.y, lower_lower.z), v))
                * factor.x) * (1 - factor.y)
        + ((vs((uint3)(lower.x, upper.y, upper_lower.z), v)
                - vs((uint3)(lower.x,upper.y, lower_lower.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, upper.y, upper_lower.z), v)
                        - vs((uint3)(upper.x, upper.y, lower_lower.z), v))
                        * factor.x) * factor.y) * (1 - factor.z)
        + (((vs((uint3)(lower.x, lower.y, upper_upper.z), v)
                - vs((uint3)(lower.x, lower.y, lower upper.z), v))
                * (1 - factor.x)
                + (vs((uint3)(upper.x, lower.y, upper_upper.z), v)
                        - vs((uint3)(upper.x, lower.y, lower upper.z), v))
                        * factor.x) * (1 - factor.y)
                + ((vs((uint3)(lower.x, upper.y, upper_upper.z), v)
                        - vs((uint3)(lower.x, upper.y, lower_upper.z), v))
                        * (1 - factor.x)
                        + (vs((uint3)(upper.x, upper.y, upper upper.z), v)
                                - vs(
                                        (uint3)(upper.x, upper.y,
                                                lower_upper.z), v))
```

```
                                    * factor.x) * factor.y) * factor.z;

    return gradient
            * (float3)(v.dim.x / v.size.x, v.dim.y / v.size.y,
                    v.dim.z / v.size.z) * (0.5f * 0.00003051944088f);
}

inline float3 get_translation(  __global const  Matrix4* restrict view) {
    return (float3)(view->data[0].w, view->data[1].w, view->data[2].w);
}


inline float3 myrotate(     __global const  Matrix4* restrict M, const float3 v) {
    return (float3)(dot((float3)(M->data[0].x, M->data[0].y, M->data[0].z), v),
            dot((float3)(M->data[1].x, M->data[1].y, M->data[1].z), v),
            dot((float3)(M->data[2].x, M->data[2].y, M->data[2].z), v));
}

float4 raycast(const Volume v, const uint2 pos, __global  const Matrix4* view,
        const float nearPlane, const float farPlane, const float step,
        const float largestep) {

    const float3 origin = get_translation(view);
    const float3 direction = myrotate(view, (float3)(pos.x, pos.y, 1.f));

    // intersect ray with a box
    //
    // www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm
    // compute intersection of ray with all six bbox planes
    const float3 invR = (float3)(1.0f) / direction;
    const float3 tbot = (float3) - 1 * invR * origin;
    const float3 ttop = invR * (v.dim - origin);

    // re-order intersections to find smallest and largest on each axis
    const float3 tmin = fmin(ttop, tbot);
    const float3 tmax = fmax(ttop, tbot);

    // find the largest tmin and the smallest tmax
    const float largest_tmin = fmax(fmax(tmin.x, tmin.y), fmax(tmin.x, tmin.z));
    const float smallest_tmax = fmin(fmin(tmax.x, tmax.y),
            fmin(tmax.x, tmax.z));

    // check against near and far plane
    const float tnear = fmax(largest_tmin, nearPlane);
    const float tfar = fmin(smallest_tmax, farPlane);

    if (tnear < tfar) {
        // first walk with largesteps until we found a hit
        float t = tnear;
        float stepsize = largestep;
        float f_t = interp(origin + direction * t, v);
        float f_tt = 0;
        if (f_t > 0) { // ups, if we were already in it, then don't render anything here
            for (; t < tfar; t += stepsize) {
                f_tt = interp(origin + direction * t, v);
                if (f_tt < 0)                  // got it, jump out of inner loop
                    break;
                if (f_tt < 0.8f)               // coming closer, reduce stepsize
                    stepsize = step;
                f_t = f_tt;
            }
            if (f_tt < 0) {           // got it, calculate accurate intersection
                t = t + stepsize * f_tt / (f_t - f_tt);
                return (float4)(origin + direction * t, t);
            }
        }
    }

    return (float4)(0);
}
__kernel void AOCrenderVolumeKernel( __global uchar * render,
```

```
        __global short2 * v_data,
        const uint3 v_size,
        const float3 v_dim,
        __global const  Matrix4* view,
        const float nearPlane,
        const float farPlane,
        const float step,
        const float largestep,
        const float3 light,
        const float3 ambient) {

    const Volume v = {v_size, v_dim,v_data};

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

    float4 hit = raycast( v, pos, view, nearPlane, farPlane,step, largestep);

    if(hit.w > 0) {
        const float3 test = as_float3(hit);
        float3 surfNorm = grad(test,v);
        if(length(surfNorm) > 0) {
            const float3 diff = normalize(light - test);
            const float dir = fmax(dot(normalize(surfNorm), diff), 0.f);
            const float3 col = clamp((float3)(dir) + ambient, 0.f, 1.f) * (float3) 255;
            vstore4((uchar4)(col.x, col.y, col.z, 0), pos.x + sizex * pos.y, render); // The
            forth value in uchar4 is padding for memory alignement and so it is for
            following uchar4
        } else {
            vstore4((uchar4)(0, 0, 0, 0), pos.x + sizex * pos.y, render);
        }
    } else {
        vstore4((uchar4)(0, 0, 0, 0), pos.x + sizex * pos.y, render);
    }

}
```

MM2meters and Bilateral Filter combined

```
__attribute__((task))
__kernel void AOCmm2_bilat(
          global float * restrict depth,
        const uint depthSize_x ,
        const uint depthSize_y ,
        const   global ushort * restrict in ,
        const uint inSize_x ,
        const int ratio,

    __global float * restrict out,
        const float mult result,
        const int r ,
    const uint2 outSize
     )
{
    for(uint pixel_y=0;pixel_y<depthSize_y;pixel_y++){
    #pragma unroll 5
        for(uint pixel_x=0;pixel_x<depthSize_x;pixel_x++){
            depth[pixel_x + depthSize_x * pixel_y] = in[pixel_x * ratio + inSize_x * pixel_y
            * ratio] / 1000.0f;
        }
    }

  const uint size_x=outSize.x;
  const uint size y=outSize.y;

  float gaussian[6]={0.8824968934,0.9692332149,1.00,0.9692332149,0.8824968934};


   for(uint pos y=0;pos y<outSize.y;pos y++){
                for(uint pos_x=0;pos_x<outSize.x;pos_x++){

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



            uint count=0;
            float t_array[25];
            float sum_array[25];


            float sum = 0.0f;
            float t = 0.0f;


            int i=-3;
            int j=-2;

            #pragma unroll
            for(int count = 0; count < 25; ++count) {

                if(count%5==0){
                  i++;
                  j=-2;
                }

                    const uint2 curPos = (uint2)(clamp(pos_x + i, 0u, size_x-1), clamp(pos_y
                    + j, 0u, size_y-1));
                    const float curPix = depth[curPos.x + curPos.y * size_x];
                    if(curPix > 0) {
                        const float mod = (curPix - center)* (curPix - center);
                        const float factor = gaussian[i + r] * gaussian[j + r] * exp(-mod /
                        mult result);
                        t_array[count] = factor * curPix;
                        sum_array [count] = factor;
                    }
                j++;
            }

           //removing loop dependency in sum and t
           #pragma unroll
```

```
            for(int j=0;j<25;j++){
              t += t_array[j];
              sum += sum array [j];
            }

             out[pos x + size x * pos y] = t / sum;


            //To remove the loop caried dependency, this if statement has been moved at the
            end of the kernel
             if ( center == 0 ) {
             out[pos_x + size_x * pos_y] = 0;
             }
        }

    }

}
```

Depth2Vertex and Vertex2Normal combined

```
typedef struct sMatrix4 {
    float4 data[4];
} Matrix4;



int find_min(int a,int b){

  int min=a;
  if(a>b){
    min=b;
  }
  return min;
}

int find_max(int a,int b){

  int max=a;
  if(b>a){
    max=b;
  }
  return max;
}


inline float3 myrotate(__global const Matrix4 * restrict M, const float3 v) {
    return (float3)(dot((float3)(M->data[0].x, M->data[0].y, M->data[0].z), v),
            dot((float3)(M->data[1].x, M->data[1].y, M->data[1].z), v),
            dot((float3)(M->data[2].x, M->data[2].y, M->data[2].z), v));
}


#define INVALID -2


  kernel void AOC d2v v2n(   global float * restrict vertex, // float3
        const uint2 vertexSize ,
        const __global float * restrict depth,
        const uint2 depthSize ,
        __global const  Matrix4* restrict invK,
    const uint2 outSize ,

    __global float * restrict normal,    // float3
        const uint2 normalSize
    ) {

  float3 vert= (float3)  (outSize.x,outSize.y,1.0f);


  for(int pixel_y=0;pixel_y<outSize.y;pixel_y++){

    #pragma unroll 10
    for(int pixel_x=0;pixel_x<outSize.x;pixel_x++){

        if(pixel x < depthSize.x || pixel y < depthSize.y ) {

        float3 res = (float3) (0);

        if(depth[pixel_x + depthSize.x * pixel_y] > 0) {
            res = depth[pixel x + depthSize.x * pixel y] * (myrotate(invK, (float3)(pixel x,
            pixel_y, 1.f)));
        }

         vstore3(res, pixel_x + vertexSize.x * pixel_y,vertex);     // vertex[pixel] =
         //vertex[pixel_x + pixel_y * outSize.x]=res;
       }
    }
  }


  //vertext to normal
```

```
  for(int pixel_y=0;pixel_y<outSize.y;pixel_y++){

    for(int pixel_x=0;pixel_x<outSize.x;pixel_x++){

            if(pixel x < vertexSize.x || pixel y < vertexSize.y ){

            uint2 vleft = (uint2)(find_max((int)(pixel_x)-1,0), pixel_y);
            uint2 vright = (uint2)(find_min(pixel_x+1,vertexSize.x-1), pixel_y);
            uint2 vup = (uint2)(pixel_x, find_max((int)(pixel_y)-1,0));
            uint2 vdown = (uint2)(pixel x, find min(pixel y+1,vertexSize.y-1));

            const float3 left = vload3(vleft.x + vertexSize.x * vleft.y,vertex);
            const float3 right = vload3(vright.x + vertexSize.x * vright.y,vertex);
            const float3 up = vload3(vup.x + vertexSize.x * vup.y,vertex);
            const float3 down = vload3(vdown.x + vertexSize.x * vdown.y,vertex);

            if(left.z == 0 || right.z == 0|| up.z ==0 || down.z == 0) {
                vstore3((float3)(INVALID,INVALID,INVALID),pixel_x + normalSize.x *
                pixel_y,normal);
          // normal[pixel_x + normalSize.x * pixel_y] = 1;;
            }
          else{
            const float3 dxv = right - left;
            const float3 dyv = down - up;
            vstore3((float3) normalize(cross(dyv, dxv)), pixel_x + pixel_y * normalSize.x,
            normal );
             //normal[pixel_x + pixel_y * normalSize.x] = 2; // switched dx and dy to get
             factor -1
          }
      }
    }
  }


}
```