ECE 597- Stereo Vision on FPGA

Alejandro Vicente-Perez, David Hernandez

12/14/2021

Acknowledgement: I acknowledge all works including figures, codes and writings belong to me and/or persons who are referenced. I understand if any similarity in the code, comments, customized program behavior, report writings and/or figures are found, both the helper (original work) and the requestor (duplicated/modified work) will be called for academic disciplinary action.

Signature: Alejandro Perez-Vicente, David Hernandez

Abstract

This paper will outline the process and methods used to deploy stereo vision and triangulation algorithms on an FPGA. It begins with an introduction in what is Stereo Vision, depth map, and point cloud mapping. Following that, we will describe where our project could be used to overcome some current challenges. After a brief introduction we will follow with some of the methodologies, softwares, and online resources used to complete this project. Transitioning to a discussion about the results and achievements of the project. The conclusion will summarize our findings and anything that we can have interpreted from our results.

Introduction

In this project, the objective is to design a system to use Stereo Vision to create a depth map estimation and then translate that to a point cloud mapping. We will be implementing a Stereo Vision on a Field Programmable Gate Array (FPGA), utilizing a Deep Neural Network that is known as Fast and Accurate Network for Disparity Estimation (FADNet).

Description of Problem

3D Scene Reconstruction

In autonomous driving, robotics, and computer vision applications, 3D Scene reconstruction is highly relevant, whether it be exploring unknown terrain, or simply trying to drive down a busy San Francisco street. 3D scene reconstruction aids us in being able to determine where objects are and how they interact with the environment we are currently in. Some of the things that are relevant to think about are the relative distance of objects and if those objects may be approaching you, some of the technologies that have been used are radar and lidar in many applications for the military. Advancements in hardware accelerators and image processing methodologies have made it possible for cameras to be the sensor that aids us in understanding our environment visually.

Hardware Acceleration

Often when algorithms are created some of the main goals would be optimization and resource management especially when it comes to Machine Learning. When implementing these algorithms on embedded platforms both memory resources and computing power become limited. Therefore we try to come up with algorithms that can minimize utilization of those resources while maximizing the performance.

Methods Used

A brief outline of what will be described below. The methods are broken down into 3 sections: Hardware, Software, and Stereo Vision/Machine Learning. Just to get a brief overview of all they are all related is Vivado & Vitis AI are software tools which we use to build our hardware such as the Deep Learning Unit, and then we implement stereo vision and machine learning networks such as FADNet onto the FPGA using Vitis AI.

Software

Vivado

Vivado is a design tool that we can use to build block designs as well as write HDL code for our FPGAs. Allowing for simulation, synthesis, implementation, place & route, and writing bitstream. An industry standard tool that integrates well with other tools also made by Xilinx such as Vitis and Vitis AI which will be discussed more in the following section. In our project we used this to generate the block for our structure. Later on we'll showcase some of the block designs for our project.

Vitis AI

Vitis AI is a development environment made to be able to develop and deploy AI models on to Xilinx hardware. Consisting of many frameworks, models, and the DPU. All of them allow you to create and deploy networks along with viewing examples. In our project we used the model zoo which is essentially a collection of pre-trained neural networks to be able to deploy a model onto our hardware with ease, and not worry about training.

Hardware(Need to work on)

Xilinx KV260 Kria Board

This is a board sold by Xilinx and contains a Zynq® UltraScale+™ MPSoC, upgrading from the previous Zynq-7000. The MPSoC contains Dual and Quad core Cortex-A53 64-bit processors. There are some Linux OS options such as PetaLinux 2021.1 which allows us to obtain everything to build and deploy embedded applications on this board. The board itself gas many interfaces and connectors allowing us to connect more hardware for future advancement and iterations of this project.

Deep Learning Unit (DPU)

The DPU has been optimized for convolutional neural networks (discussed later on). The DPU contains some configurable parameters for users to be able to optimize their resources and customize different features for their own applications. This IP is able to be implemented on a Zynq®-7000 SoC or Zynq® UltraScale+™ MPSoC. With the DPU their a re a couple things that are also necessary such as an Application Processing Unit, Processing Engine, and RAM all helping coordinate instructions, transfer data, and service interrupts.

Machine Learning & Computer Vision

Below will describe a brief background on Machine Learning and how it works and we will then tie this into stereo vision since this is prerequisite knowledge to have before discussing the algorithm described below.

Neural Networks (NN & DNN)

Neural networks mimic the way the human brain works and this is done through following an algorithm. To the right is a visual representation of a neural network and we will also showcase the mathematical model. What can be seen in Fig. 1 is each circle is what is known as a neuron which will hold a value that is determined by weights and biases. Eq. 1 essentially shows us how the weights and biases will affect our value for the neuron.

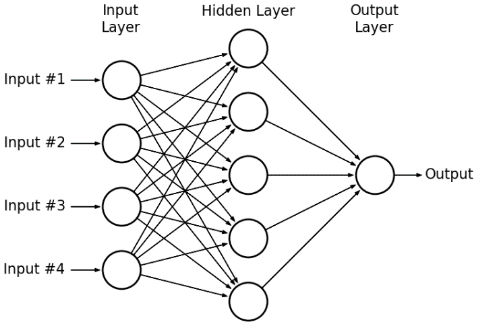


Figure 1. Neural Network Diagram

$$\sum_{i=1}^{m} = w_i x_i + bias = w_1 x_1 + w_2 x_2 + ... + w_m x_m + bias$$

Equation 1. Mathematical Expression for Neural Networks

A deep neural network is essentially just a neural network with more than ~3 layers. With more layers come more weight and biases, leading to higher computational complexity. This is often a concern when working with these types of neural networks.

Training

One of the most common ways of training neural networks is through backpropagation. This algorithm uses the mathematical property of the gradient, what backpropagation is it wants to minimize a ‘cost function’ which tells us how off it is from the actual prediction. Since our goal is to try to be highly accurate and therefore we will use gradient descent. For example if we had weight and biases: a,b,c,d,e,f and J be equal to the cross entropy function J. The gradients would be as follows:

$$\frac{\partial J}{\partial a}, \frac{\partial J}{\partial b}, \frac{\partial J}{\partial c}, \frac{\partial J}{\partial d}, \frac{\partial J}{\partial e}, \frac{\partial J}{\partial f},$$

Figure 2: Gradient of Parameters

$$\theta_{i+1} = \theta_i - \epsilon \frac{\partial J}{\partial \theta}$$

Equation 2: Stochastic Gradient Descent

The Eq. 2 is how we update our weights and biases if they are contained in a set known as θ.

Computer Vision & Stereo Vision

Computer Vision essentially boils down to being able to gain a level of understanding from images that is high. The process is as follows: acquire, process, analyze, understand. This is from a high level. Stereo vision is within the category of computer vision and it follows the same process but has some more specific techniques. The technique is as follows: there are two cameras that are separated by an accurately known distance, and both capture the image at the same time. When you look at those images side by side there is found to be a slight difference, that is known as the disparity. What can be constructed is a disparity map to show the apparent difference or sometimes known as ‘motion’. The disparity map can then be used to calculate a depth map which follows an Eq. 3, most parameters will be known before-hand such as baseline & focal length.

$$Depth = \frac{Baseline * focal\ length}{Disparity}$$

Equation. 3 Depth Equation

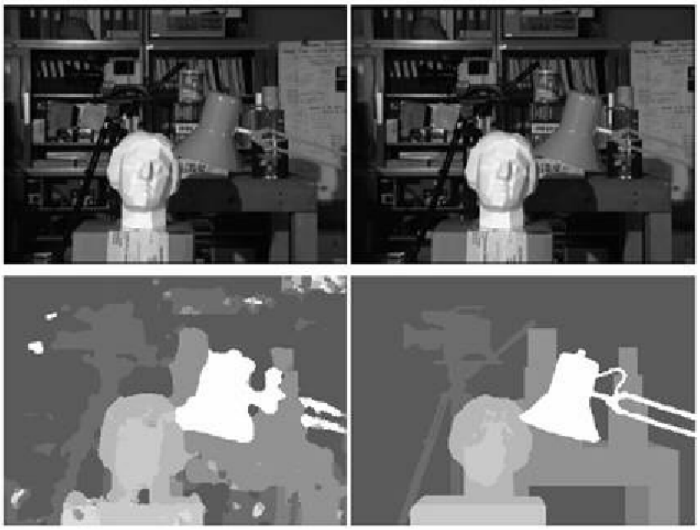


Figure 3: Stereo Images, Disparity Map, and Depth Map [12]

Above can be seen an example of expected results of this stereo vision algorithm.

Convolutional Neural Networks

Convolutional Neural Networks (CNN) are essentially a combination of deep learning and computer vision. What is the main difference between this approach and regular neural networks is that CNN's use the convolutional operation to extract features from images.

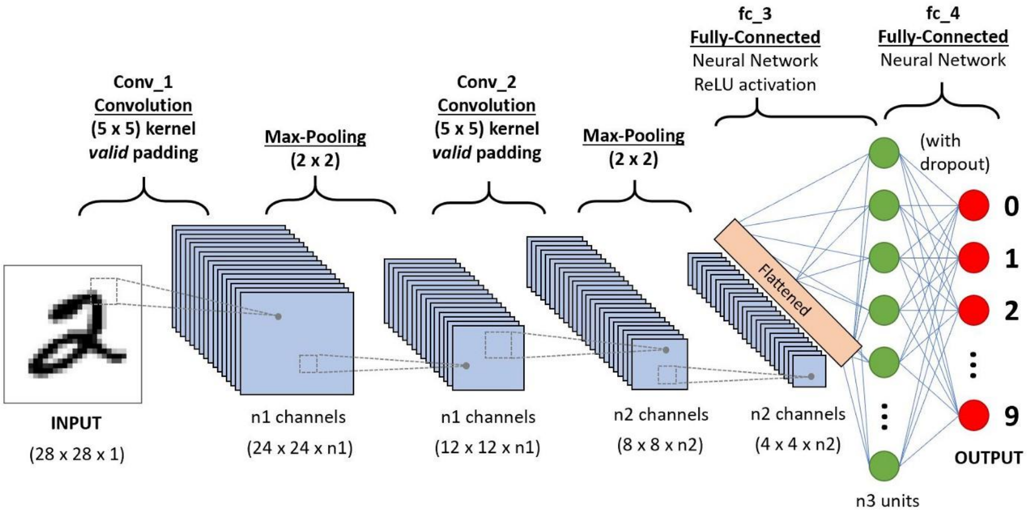


Figure 4: Convolutional Neural Network Example [13]

There are two parts to this process, one being feature extraction and the other is classification. Most of the layers in a Convolutional Neural Network try to break up the image into pieces through using a smaller size filter that will have a number of strides to go across an image, this is to be able to break up the image into smaller pieces. There are a couple of common functions: ReLU (Rectified Linear Unit), Pooling, and Flatten, each of them attempts to reduce spatial complexity by reducing sets or limiting set space. Rectified Linear Unit which essentially follows the behavior of this mathematical equation Eq. 3. This essentially limits the space to only positive real numbers therefore reducing computational complexity.

$$g(x) \ = \ max(0, x)$$

Equation 3: ReLU Equation

Pooling attempts to reduce the spatial size by having a set of pixels and either applying max pooling which will take the max value, or average pooling and will take the average of the set of pixels. Flatten just takes the set of matrices and 'flattens' them into a column vector, aiding this to be an input into a neural network for classification, operating similar to the neural networks described above

Fast and Accurate Disparity Network

This Network is based off of a Deep neural network architecture for disparity estimation, the goal was to be able to compete with the prediction accuracy of other models such as the CVM-Conv3D but having better overall performance. We attempted to implement this network into our project and all of the information above was a prerequisite to be able to understand the following.

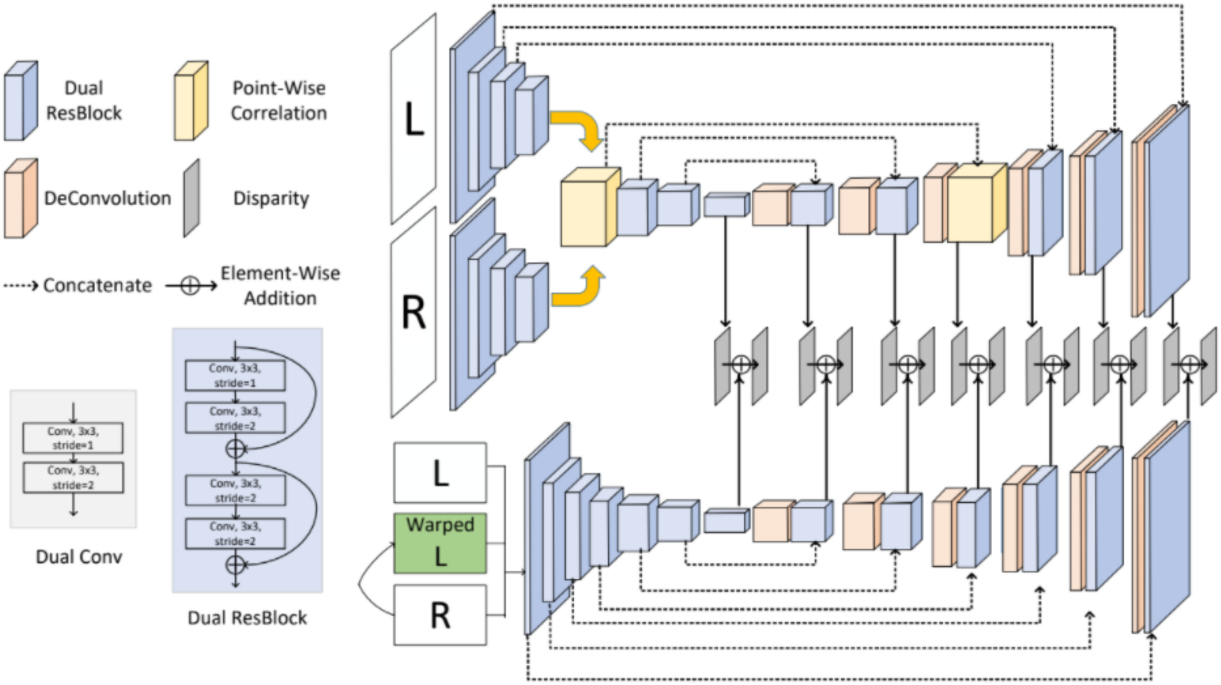


Figure 5: The Structure of FADNet

The structure of FADNet is based off DispNetC and DispNetS which use an encoder-decoder structure but there are some differences made in FADNet. The Dual-Conv modules shrink the feature map size by half, there is also a correlation layer to reduce the end-points error. The residual block is used for image classification tasks which are used to train very deep neural networks and learn robust features[FADNet Article]. The Dual Residual Block allows us to have very deep models without increasing training difficulty along with allowing us to increase feature extraction and down-sampling layers from 5 to 7 [FADNet Article].

Implementation

DPU Block Design

The implementation of a Vitis platform to the embedded OS requires the following files: the workflow required for creating a full working application platform on a configurable Xilinx device requires the usage of Vitis, Vivado and Petalinux software tools. The process starts by describing a hardware architecture design in Vivado[1]. For our purpose, the hardware design will be composed mainly of the DPUCZX8G, a programmable engine optimized for convolutional neural networks. It is composed of a high performance scheduler module, a hybrid computing array module, an instruction fetch unit module, and a global memory pool module. The DPUCZDX8G IP block [2] can be inserted on the Programmable Logic (PL) by using the Vivado Block Design with the right connections to the Processing System (PS). Below we can see the different modules for the correct functionality of the IP block.

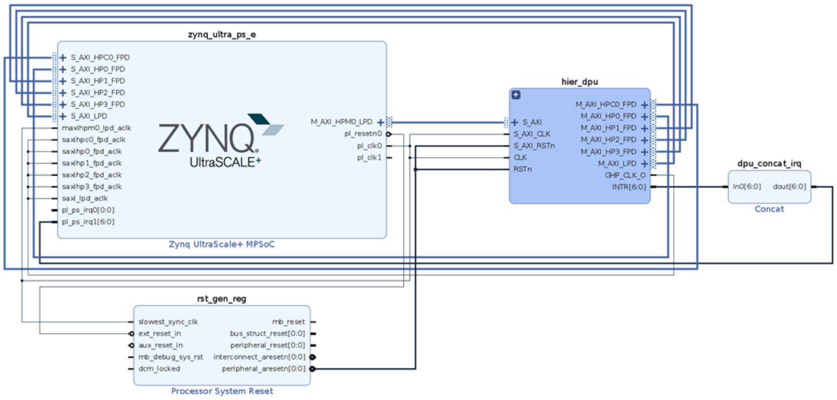


Figure 6: Block Design for Whole System

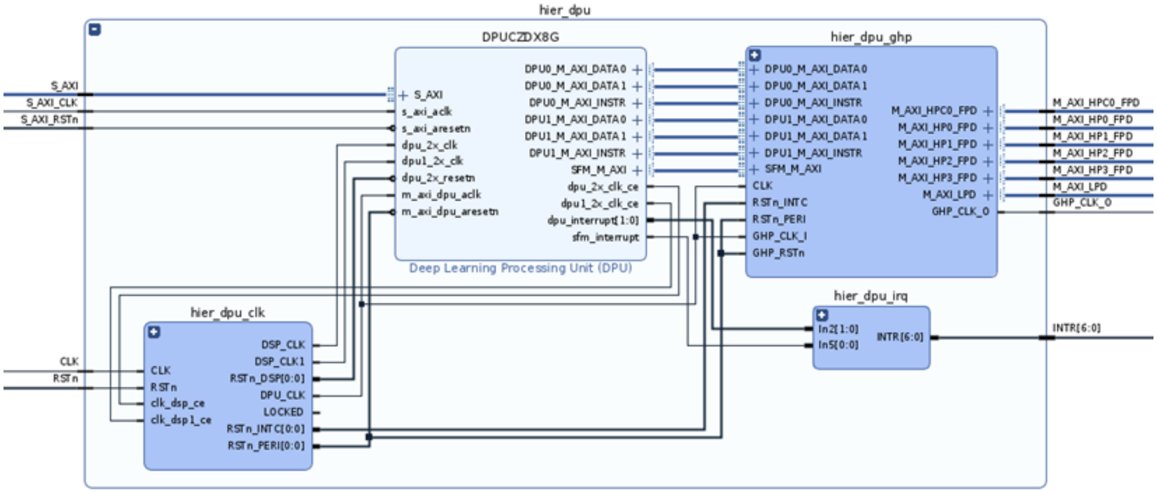
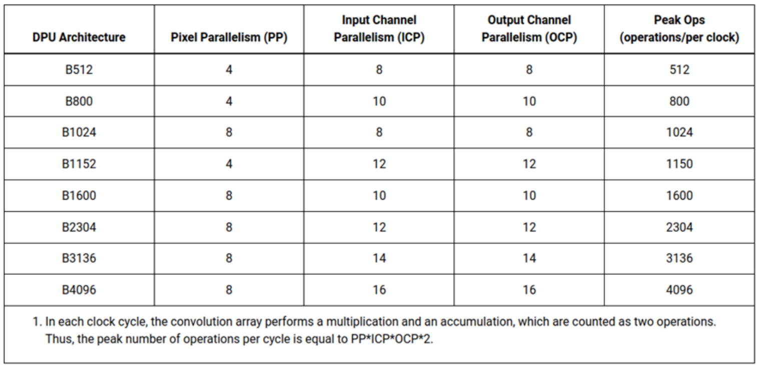


Figure 7: Block Design for DPU IP Block

The DPUCZX8G internals cannot be accessed as it is Xilinx intellectual property design, however it can be modified and customized for different performance specifications. The DPU IP can be configured with various optimized architectures related to the parallelism of the convolution unit.

There are three dimensions of parallelism in the DPU convolution architecture: pixel parallelism, input channel parallelism, and output channel parallelism. The input channel parallelism is always equal to the output channel parallelism (this is equivalent to channel_parallel in the previous table). The different architectures require different programmable logic resources. The larger architectures can achieve higher performance with more resources. The parallelism for the different architectures is listed in the following table.

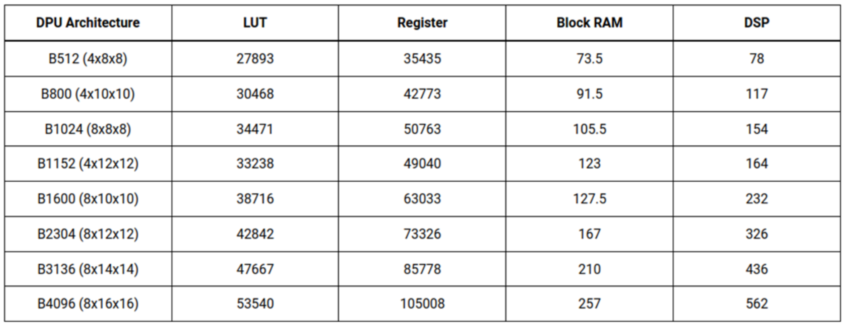


DPU Architecture	Pixel Parallelism (PP)	Input Channel Parallelism (ICP)	Output Channel Parallelism (OCP)	Peak Ops (operations/per clock)
B512	4	8	8	512
B800	4	10	10	800
B1024	8	8	8	1024
B1152	4	12	12	1150
B1600	8	10	10	1600
B2304	8	12	12	2304
B3136	8	14	14	3136
B4096	8	16	16	4096

1. In each clock cycle, the convolution array performs a multiplication and an accumulation, which are counted as two operations. Thus, the peak number of operations per cycle is equal to PP*ICP*OCP*2.

Table 1:

When porting the DPU to the Vivado block design, the IP can be modified with a include header file called "dpu_conf.vh" where you specify the architecture size and also the resource optimizations such as low RAM usage, depthwise convolutions, average pooling optimizations and other layer fusion implementations.



DPU Architecture	LUT	Register	Block RAM	DSP
B512 (4x8x8)	27893	35435	73.5	78
B800 (4x10x10)	30468	42773	91.5	117
B1024 (8x8x8)	34471	50763	105.5	154
B1152 (4x12x12)	33238	49040	123	164
B1600 (8x10x10)	38716	63033	127.5	232
B2304 (8x12x12)	42842	73326	167	326
B3136 (8x14x14)	47667	85778	210	436
B4096 (8x16x16)	53540	105008	257	562

Table 2:

Afterwards we generate the Vivado hardware design, the next step is to synthesize the design and generate a bitstream file, which will be altogether saved on a Xilinx Support Archive or .xsa extension file. With the .xsa file you generate a hardware description platform for the Vitis platform generator. The Vitis platform generator [3] will create a binary file containing the bitstream ready for the Vitis platform configuration on the FPGA hardware. Also a ``.xclbin" file is generated, unlike a common bitstream file generated by the Vivado synthesis tool, the xclbin file is designed so that the whole customized platform can be loaded during run-time without requiring rebooting the whole system. This feature is very important specially when deploying an embedded system which requires to perform different tasks, by having a software capable of re-configuring the hardware platform with just loading and unloading required firmware components[4]. Lastly it is necessary to create a device tree structure file. This file is responsible for describing all the hardware drivers and components that will be required for building the application on the embedded linux kernel.

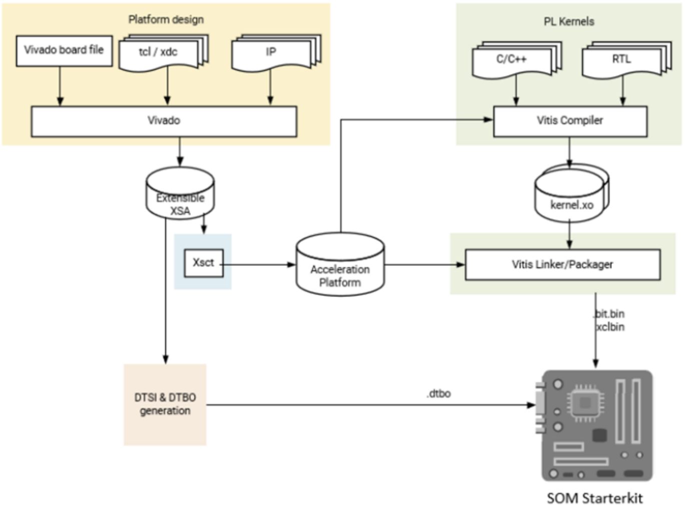
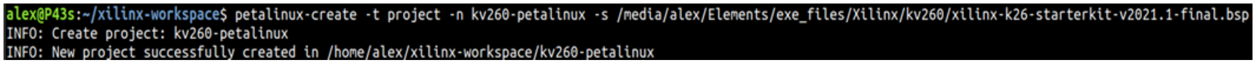


Figure 8: Hardware Platform Design Workflow

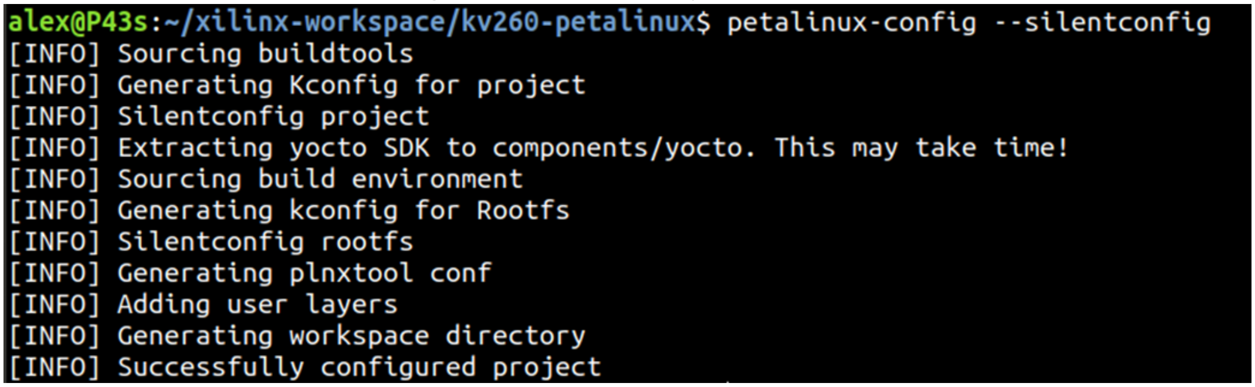
Building Petalinux Firmware

First, we create a petalinux project with the Board Support Package (BSP) of the target FPGA, in our case it is the Kria KV260 BSP. The BSP contains all the required drivers and software layers required for building a functional embedded OS for RTOS.



```
alex@P43s:~/xilinx-workspace$ petalinux-create -t project -n kv260-petalinux -s /media/alex/Elements/exe_files/Xilinx/kv260/xilinx-k26-starterkit-v2021.1-final.bsp
INFO: Create project: kv260-petalinux
INFO: New project successfully created in /home/alex/xilinx-workspace/kv260-petalinux
```

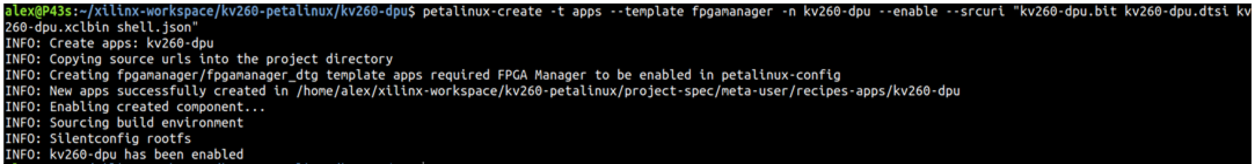
Figure 9: Petalinux Object Created



```
alex@P43s:~/xilinx-workspace/kv260-petalinux$ petalinux-config --silentconfig
[INFO] Sourcing buildtools
[INFO] Generating Kconfig for project
[INFO] Silentconfig project
[INFO] Extracting yocto SDK to components/yocto. This may take time!
[INFO] Sourcing build environment
[INFO] Generating kconfig for Rootfs
[INFO] Silentconfig rootfs
[INFO] Generating plnxtool conf
[INFO] Adding user layers
[INFO] Generating workspace directory
[INFO] Successfully configured project
```

Figure 10: Configure Petalinux Project for Silent Mode to simplify the process

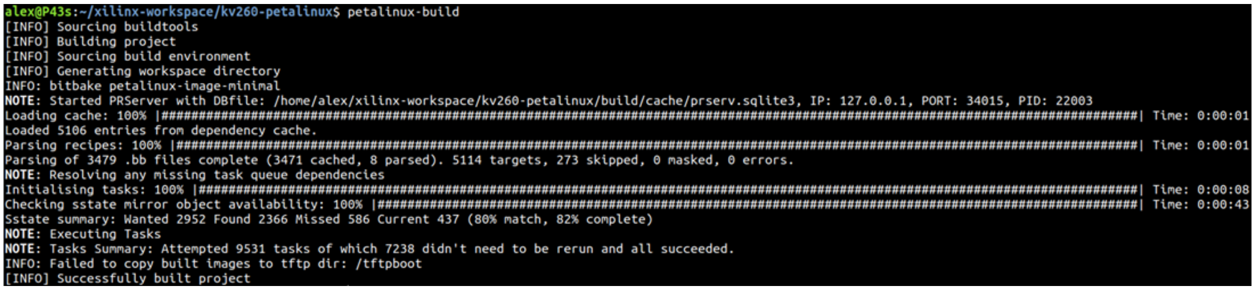
For compiling our customized hardware platform inside the pelatinux project, we gather the device tree structure file, the bit file with the PL bitstream and the xclbin file for platform management integration on the RTOS.



```
alex@P43s:~/xilinx-workspace/kv260-petalinux/kv260-dpu$ petalinux-create -t apps --template fpgamanager -n kv260-dpu --enable --srcuri "kv260-dpu.bit kv260-dpu.dtsi kv
260-dpu.xclbin shell.json"
INFO: Create apps: kv260-dpu
INFO: Copying source urls into the project directory
INFO: Creating fpgamanager/fpgamanager_dtg template apps required FPGA Manager to be enabled in petalinux-config
INFO: New apps successfully created in /home/alex/xilinx-workspace/kv260-petalinux/project-spec/meta-user/recipes-apps/kv260-dpu
INFO: Enabling created component...
INFO: Sourcing build environment
INFO: Silentconfig rootfs
INFO: kv260-dpu has been enabled
```

Figure 11: Enable hardware platform for petalinux project

Afterwards the petalinux project is built together with the hardware platform, generating a device tree object file. This file will be responsible for modifying the current linux kernel components for the hardware platform.



```
alex@P43s:~/xilinx-workspace/kv260-petalinux$ petalinux-build
[INFO] Sourcing buildtools
[INFO] Building project
[INFO] Sourcing build environment
[INFO] Generating workspace directory
INFO: bitbake petalinux-image-minimal
NOTE: Started PRServer with DBfile: /home/alex/xilinx-workspace/kv260-petalinux/build/cache/prserv.sqlite3, IP: 127.0.0.1, PORT: 34015, PID: 22003
Loading cache: 100% |#######################################################################################################################| Time: 0:00:01
Loaded 5106 entries from dependency cache.
Parsing recipes: 100% |#####################################################################################################################| Time: 0:00:01
Parsing of 3479 .bb files complete (3471 cached, 8 parsed). 5114 targets, 273 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |##################################################################################################################| Time: 0:00:08
Checking sstate mirror object availability: 100% |##########################################################################################| Time: 0:00:43
Sstate summary: Wanted 2952 Found 2366 Missed 586 Current 437 (80% match, 82% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 9531 tasks of which 7238 didn't need to be rerun and all succeeded.
INFO: Failed to copy built images to tftp dir: /tftpboot
[INFO] Successfully built project
```

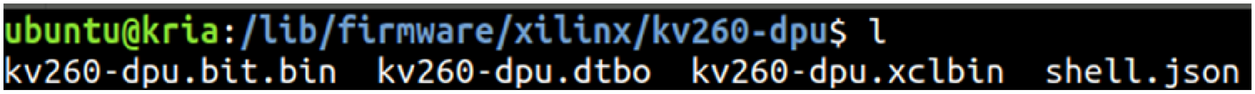
Figure 12: Build Petalinux Project

In the proposed embedded RTOS, the rootfile system used for running the software applications was a lightweight Ubuntu 20.04 distribution optimized for aarch64 architecture. Since our project will require the deployment of neural networks inside the DPUCZDX8G accelerator, the Vitis AI library components [6] will be needed for the edge platform. This library contains C++ API with optimized implementations frequently used for the deployment on neural networks from edge to cloud devices. The Vitis AI library will be required for launching the deep learning compute kernels on the DPU accelerator.

DPU & FADNet Integration

The next step is to get the pre-trained model for stereo-vision reconstruction. In the proposed neural network, we used Fast Accurate Disparity Map (FadNet) neural network for performing the disparity map. This model was trained with the KITTI[7] dataset, a standard for performing benchmarks on computer vision related neural networks. Since the neural network can take weeks for training it and a lot of computing power, we extracted a pre-trained and compiled FadNet model from the “Vitis AI Model Zoo”. This is a repository database with multiple neural network models that are trained, optimized and ready for deployment on FPGA targets. The compiled model is denoted as an intermediate representation binary or Xilinx Intermediate Representation (XIR) [9] with “.xmodel” extension. This “.xmodel” binaries include the kernel functions that define all operation layers of the neural network architecture. For the FadNet model, it will be subdivided into three intermediate representation binaries, two for the kernel computations on each stereo-pair image(left and right) and one last for the subsequent point-wise correlation and disparity estimation.

Before deploying any neural network model, the DPUCZDX8G hardware platform must be loaded into the embedded linux for re-configuring the FPGA. After booting and logging into the MPSoC OS, we will use “xlnx-config” [8] application manager. This program helps managing, installing and updating the FPGA firmware during runtime. The firmware and hardware platform bitstream files must be under the **/lib/firmware/xilinx** folder.

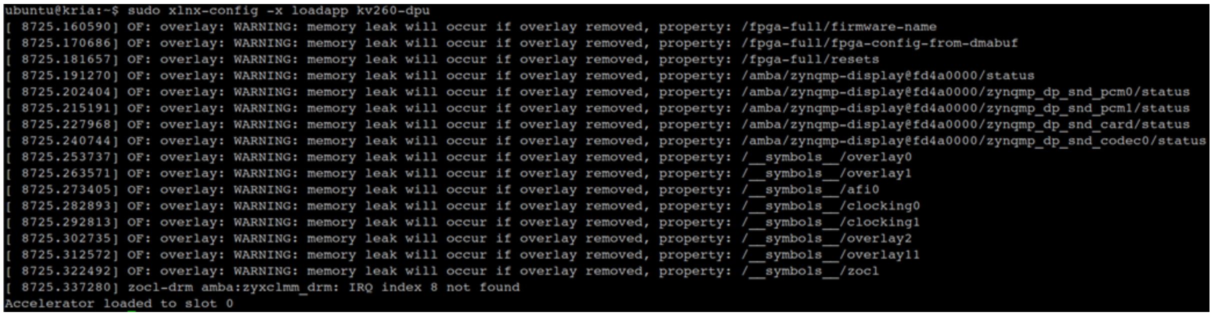


```
ubuntu@kria:/lib/firmware/xilinx/kv260-dpu$ l
kv260-dpu.bit.bin  kv260-dpu.dtbo  kv260-dpu.xclbin  shell.json
```

Figure 13: Device tree overlay and xclbin hardware platform binary

Inside the hardware platform folder there should be the bitstream file for the PL, the xclbin binary for the hardware platform configuration and the device tree overlay for adding the necessary drivers to the FPGA board .

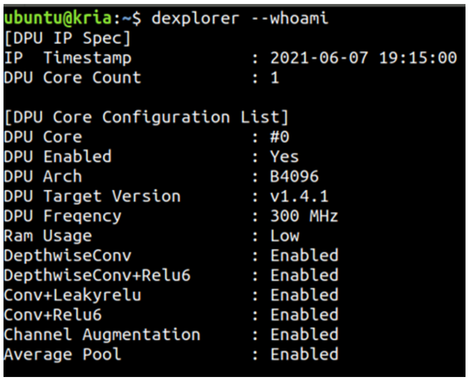
To load the hardware platform into the OS type the following command:



```
ubuntu@kria:~$ sudo xlnx-config -x loadapp kv260-dpu
[ 8725.160590] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /fpga-full/firmware-name
[ 8725.170686] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /fpga-full/fpga-config-from-dmabuf
[ 8725.181657] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /fpga-full/resets
[ 8725.191270] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /amba/zynqmp-display@fd4a0000/status
[ 8725.202404] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /amba/zynqmp-display@fd4a0000/zynqmp_dp_snd_pcm0/status
[ 8725.215191] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /amba/zynqmp-display@fd4a0000/zynqmp_dp_snd_pcm1/status
[ 8725.227968] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /amba/zynqmp-display@fd4a0000/zynqmp_dp_snd_card/status
[ 8725.240744] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /amba/zynqmp-display@fd4a0000/zynqmp_dp_snd_codec0/status
[ 8725.253737] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/overlay0
[ 8725.263571] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/overlay1
[ 8725.273405] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/afi0
[ 8725.282893] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/clocking0
[ 8725.292813] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/clocking1
[ 8725.302735] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/overlay2
[ 8725.312572] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/overlay11
[ 8725.322492] OF: overlay: WARNING: memory leak will occur if overlay removed, property: /__symbols__/zocl
[ 8725.337280] zocl-drm amba:zyxclmm_drm: IRQ index 8 not found
Accelerator loaded to slot 0
```

Figure 14: Load Hardware Platform DPU to FPGA board

To verify the firmware and xclbin hardware platform was successfully loaded into the OS, use the "Dexplorer" utility command for assuring the correct DPU integration into the FPGA hardware

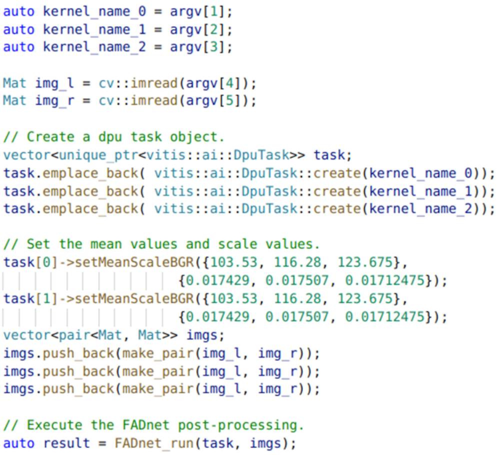


```
ubuntu@kria:~$ dexplorer --whoami
[DPU IP Spec]
IP  Timestamp            : 2021-06-07 19:15:00
DPU Core Count           : 1

[DPU Core Configuration List]
DPU Core                 : #0
DPU Enabled              : Yes
DPU Arch                 : B4096
DPU Target Version       : v1.4.1
DPU Freqency             : 300 MHz
Ram Usage                : Low
DepthwiseConv            : Enabled
DepthwiseConv+Relu6      : Enabled
Conv+Leakyrelu           : Enabled
Conv+Relu6               : Enabled
Channel Augmentation     : Enabled
Average Pool             : Enabled
```

Figure 15: Loaded DPU specifications

Once the DPU hardware platform is fully loaded into the device, the following step consists in writing a C++ executable file for the full stereo-image processing pipeline, from receiving images to organizing the sequence of kernel computations.

First the program gets as inputs the three xmodel binaries with all the kernel operations that define the neural network. The stereo-pair images are stored in an OpenCV [10] Matrix object. Afterwards comes the definition of three DPU task objects, which are responsible for all the execution and data management of each kernel computation.

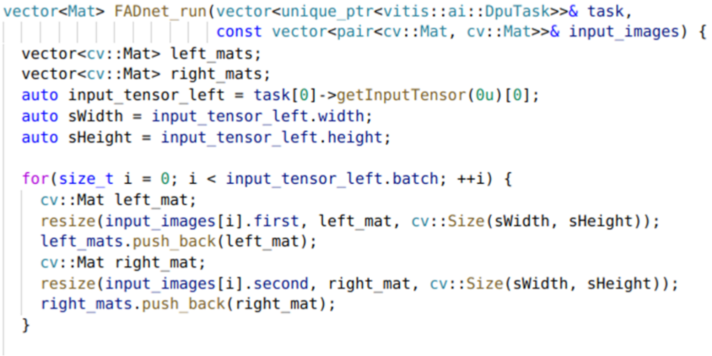


```
auto kernel_name_0 = argv[1];
auto kernel_name_1 = argv[2];
auto kernel_name_2 = argv[3];

Mat img_l = cv::imread(argv[4]);
Mat img_r = cv::imread(argv[5]);

// Create a dpu task object.
vector<unique_ptr<vitis::ai::DpuTask>> task;
task.emplace_back( vitis::ai::DpuTask::create(kernel_name_0));
task.emplace_back( vitis::ai::DpuTask::create(kernel_name_1));
task.emplace_back( vitis::ai::DpuTask::create(kernel_name_2));

// Set the mean values and scale values.
task[0]->setMeanScaleBGR({103.53, 116.28, 123.675},
                         {0.017429, 0.017507, 0.01712475});
task[1]->setMeanScaleBGR({103.53, 116.28, 123.675},
                         {0.017429, 0.017507, 0.01712475});
vector<pair<Mat, Mat>> imgs;
imgs.push_back(make_pair(img_l, img_r));
imgs.push_back(make_pair(img_l, img_r));
imgs.push_back(make_pair(img_l, img_r));

// Execute the FADnet post-processing.
auto result = FADnet_run(task, imgs);
```

Figure 16: Image Capture and DpU task creation

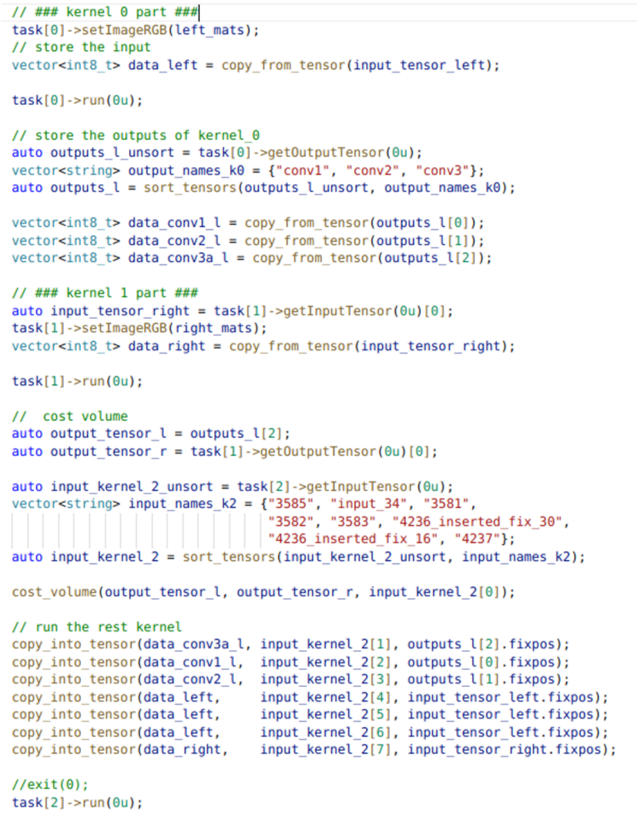
Kernel Deployment

It is recommended that both images should pass through a scale normalization for a better pixel quality and avoid disparity distortion. The FadNet processing consists of feeding the pre-processed image tensors across the different kernel tasks.



```cpp
vector<Mat> FADnet_run(vector<unique_ptr<vitis::ai::DpuTask>>& task,
                       const vector<pair<cv::Mat, cv::Mat>>& input_images) {
  vector<cv::Mat> left_mats;
  vector<cv::Mat> right_mats;
  auto input_tensor_left = task[0]->getInputTensor(0u)[0];
  auto sWidth = input_tensor_left.width;
  auto sHeight = input_tensor_left.height;

  for(size_t i = 0; i < input_tensor_left.batch; ++i) {
    cv::Mat left_mat;
    resize(input_images[i].first, left_mat, cv::Size(sWidth, sHeight));
    left_mats.push_back(left_mat);
    cv::Mat right_mat;
    resize(input_images[i].second, right_mat, cv::Size(sWidth, sHeight));
    right_mats.push_back(right_mat);
  }
```

Figure 17: Stereo-pair images resizing for kernel computation

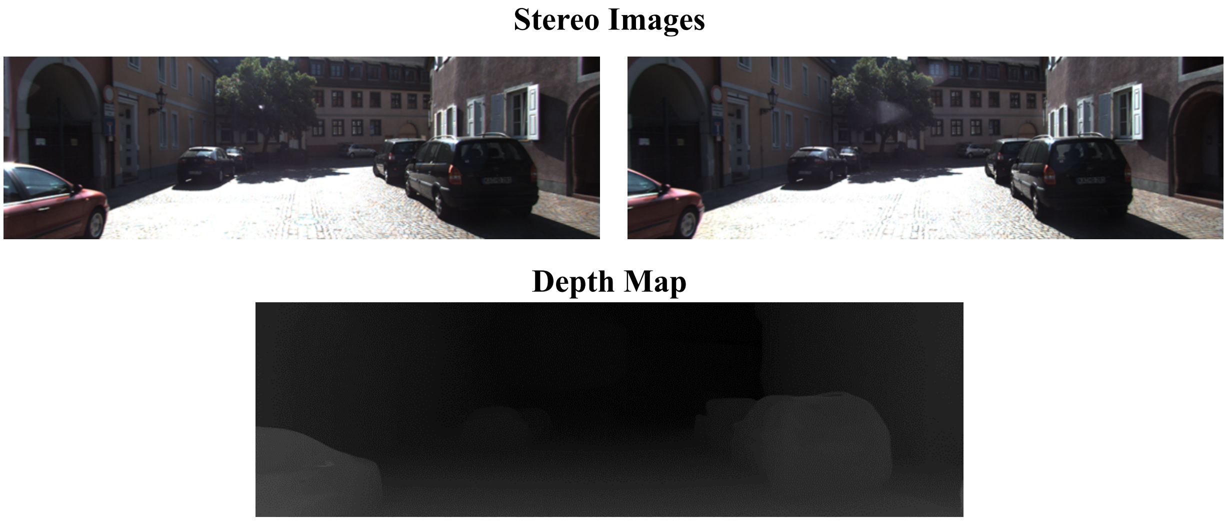


```cpp
// ### kernel 0 part ###
task[0]->setImageRGB(left_mats);
// store the input
vector<int8_t> data_left = copy_from_tensor(input_tensor_left);

task[0]->run(0u);

// store the outputs of kernel_0
auto outputs_l_unsort = task[0]->getOutputTensor(0u);
vector<string> output_names_k0 = {"conv1", "conv2", "conv3"};
auto outputs_l = sort_tensors(outputs_l_unsort, output_names_k0);

vector<int8_t> data_conv1_l = copy_from_tensor(outputs_l[0]);
vector<int8_t> data_conv2_l = copy_from_tensor(outputs_l[1]);
vector<int8_t> data_conv3a_l = copy_from_tensor(outputs_l[2]);

// ### kernel 1 part ###
auto input_tensor_right = task[1]->getInputTensor(0u)[0];
task[1]->setImageRGB(right_mats);
vector<int8_t> data_right = copy_from_tensor(input_tensor_right);

task[1]->run(0u);

//  cost volume
auto output_tensor_l = outputs_l[2];
auto output_tensor_r = task[1]->getOutputTensor(0u)[0];

auto input_kernel_2_unsort = task[2]->getInputTensor(0u);
vector<string> input_names_k2 = {"3585", "input_34", "3581",
                                 "3582", "3583", "4236_inserted_fix_30",
                                 "4236_inserted_fix_16", "4237"};
auto input_kernel_2 = sort_tensors(input_kernel_2_unsort, input_names_k2);

cost_volume(output_tensor_l, output_tensor_r, input_kernel_2[0]);

// run the rest kernel
copy_into_tensor(data_conv3a_l, input_kernel_2[1], outputs_l[2].fixpos);
copy_into_tensor(data_conv1_l,  input_kernel_2[2], outputs_l[0].fixpos);
copy_into_tensor(data_conv2_l,  input_kernel_2[3], outputs_l[1].fixpos);
copy_into_tensor(data_left,     input_kernel_2[4], input_tensor_left.fixpos);
copy_into_tensor(data_left,     input_kernel_2[5], input_tensor_left.fixpos);
copy_into_tensor(data_left,     input_kernel_2[6], input_tensor_left.fixpos);
copy_into_tensor(data_right,    input_kernel_2[7], input_tensor_right.fixpos);

//exit(0);
task[2]->run(0u);
```

Figure 18: Kernel Computations

Results & Discussion

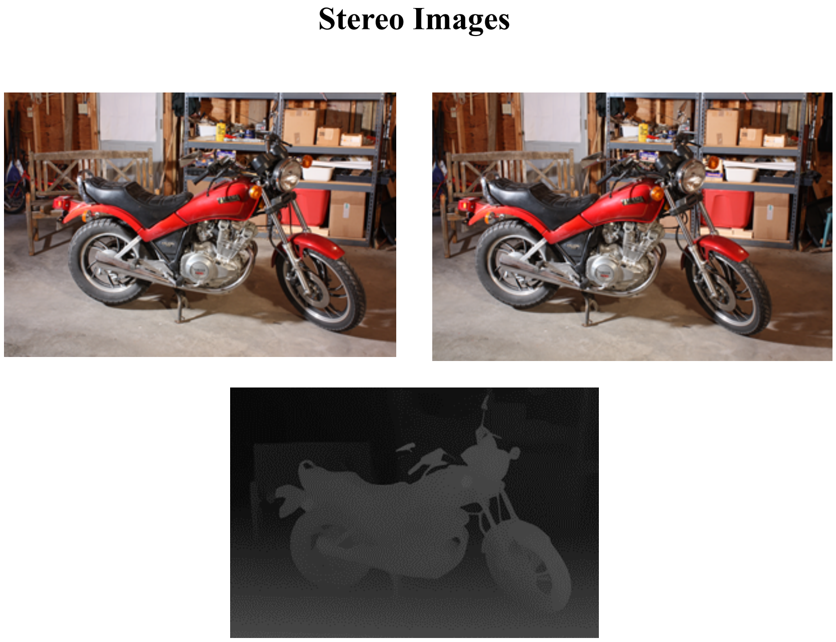
Once the kernel computations of the xmodel binaries are done, the resulting depth map is saved and plotted for verification. This model was tested with the KITTI stereo test dataset as it was trained with it. This dataset has well calibrated images for a correct testing result. Below is an example result from a stereo pair.



Stereo Images

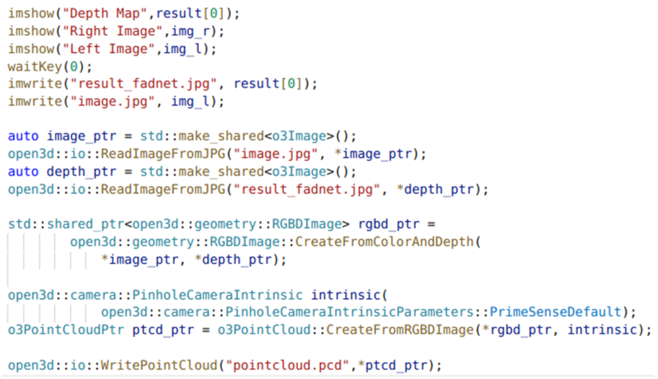
Depth Map

Figure 19: Results from FADNet



Stereo Images

Figure 20: Results from FADNet



```
imshow("Depth Map",result[0]);
imshow("Right Image",img_r);
imshow("Left Image",img_l);
waitKey(0);
imwrite("result_fadnet.jpg", result[0]);
imwrite("image.jpg", img_l);

auto image_ptr = std::make_shared<o3Image>();
open3d::io::ReadImageFromJPG("image.jpg", *image_ptr);
auto depth_ptr = std::make_shared<o3Image>();
open3d::io::ReadImageFromJPG("result_fadnet.jpg", *depth_ptr);

std::shared_ptr<open3d::geometry::RGBDImage> rgbd_ptr =
        open3d::geometry::RGBDImage::CreateFromColorAndDepth(
            *image_ptr, *depth_ptr);

open3d::camera::PinholeCameraIntrinsic intrinsic(
            open3d::camera::PinholeCameraIntrinsicParameters::PrimeSenseDefault);
o3PointCloudPtr ptcd_ptr = o3PointCloud::CreateFromRGBDImage(*rgbd_ptr, intrinsic);

open3d::io::WritePointCloud("pointcloud.pcd",*ptcd_ptr);
```

Figure 21: Depth Image to Point cloud reconstruction method

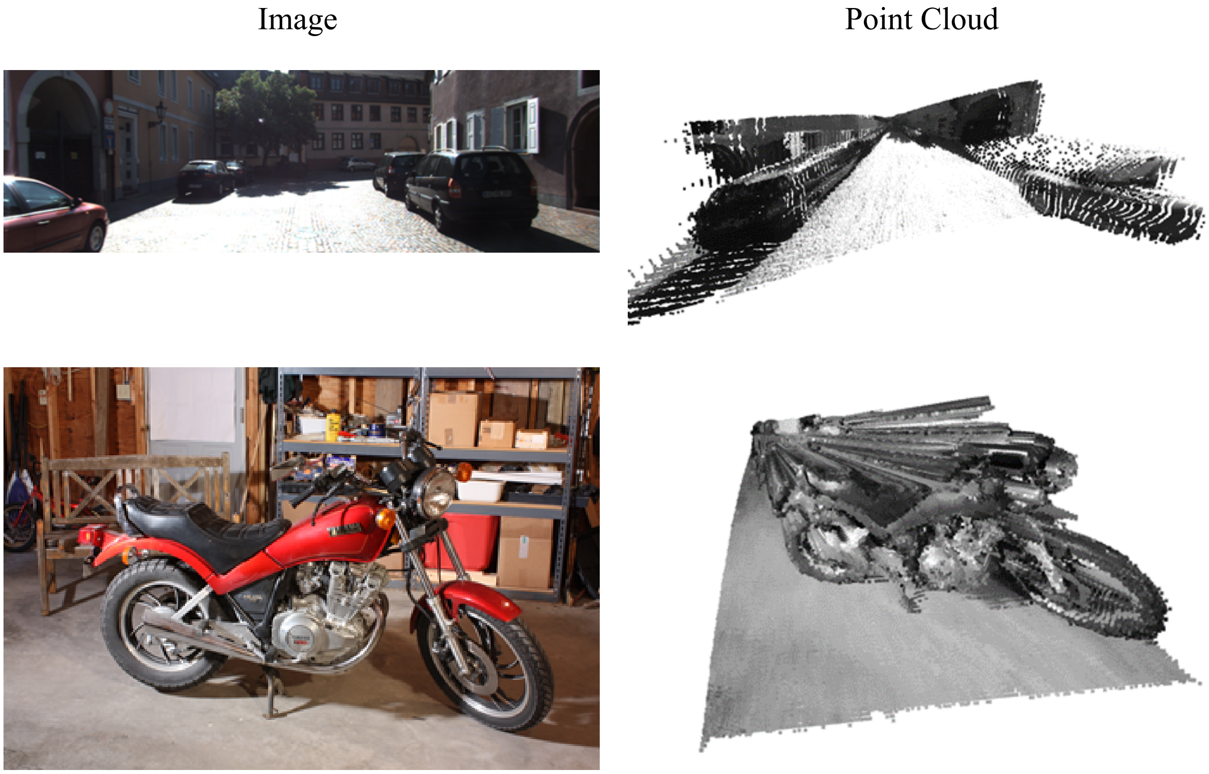


Figure 22 : Example of Image to Point Cloud

Conclusion

At this stage in our project we have come up with some good results using two frames to convert to depth map and then point cloud. Our plan for the future is to try to create a video pipeline, to create a real-time system for depth map and point cloud using two cameras. This future work will be more involved with HLS for Hardware Acceleration, along with gaining a better understanding of some of the Kria Board Architecture for data retrieval for real time processing.

References

[1] https://www.xilinx.com/developer/products/vivado.html
[2] https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/dpu_over.html
[3]https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Introduction-to-the-Vitis-Environment-for-Acceleration
[4] https://xilinx.github.io/kria-apps-docs/creating_applications/1.0/build/html/index.html
[5]https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html
[6]https://www.xilinx.com/html_docs/vitis_ai/1_1/index.html
[7]http://www.cvlibs.net/datasets/kitti/
[8]https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2057043969/Snaps+-+xlnx-config+Snap+for+Certified+Ubuntu+on+Xilinx+Devices
[9] https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/compiling_model.html
[10] https://opencv.org
[11] http://www.open3d.org/
[12] Dominguez-Morales, Manuel & Jiménez-Fernandez, Angel & Paz-Vicente, Rafael & Linares-Barranco, Alejandro & Jimenez, Gabriel. (2012). Stereo Matching: From the Basis to Neuromorphic Engineering. 10.5772/45901.
[13]https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53