**Hand-Gesture Based Actuation Using Computer Vision**



**Final Year Project Report**

**Presented**

**by**

**Ali Arslan**

CIIT/FA17-ECE-004/ISB

**Usama Mustafa**

CIIT/FA17-ECE-042/ISB

**Raja Nofil Hassan**

CIIT/FA17-ECE-033/ISB

**In Partial Fulfillment**

**of the Requirement for the Degree of**

***Bachelor of Science in Electrical (Electronics/Computer) Engineering***

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING**

COMSATS UNIVERSITY ISLAMABAD

Feb 2021

**Hand-Gesture Based Actuation Using Computer Vision**



**Final Year Project Report**

**Presented**

**by**

**Ali Arslan**

CIIT/FA17-ECE-004/ISB

**Usama Mustafa**

CIIT/FA17-ECE-042/ISB

**Raja Nofil Hassan**

CIIT/FA17-ECE-033/ISB

**In Partial Fulfillment**

**of the Requirement for the Degree of**

***Bachelor of Science in Electrical (Electronics/Computer) Engineering***

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING**

COMSATS UNIVERSITY ISLAMABAD

Feb 2021

***Declaration***

*We, hereby declare that this project neither as a whole nor as a part there of has been copied out from any source. It is further declared that we have developed this project and the accompanied report entirely on the basis of our personal efforts made under the sincere guidance of our supervisor. No portion of the work presented in this report has been submitted in the support of any other degree or qualification of this or any other University or Institute of learning, if found we shall stand responsible.*

**Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Name**

**Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Name**

**Signature:\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Name**

COMSATS UNIVERSITY ISLAMABAD

Feb 2021

***Hand-Gesture Based Actuation Using Computer Vision***

An Undergraduate Final Year Project Report submitted to the

Department of

**ELECTRICAL AND COMPUTER ENGINEERING**

**As a Partial Fulfillment for the award of Degree**

***Bachelor of Science in Electrical (Computer/Electronics) Engineering***

*by*

|  |  |
| --- | --- |
| **Name** | **Registration Number** |
| **Ali Arslan** | CIIT/FA17-ECE-004/ISB |
| **Usama Mustafa** | CIIT/FA17-ECE-042/ISB |
| **Raja Nofil Hassan** | CIIT/FA17-ECE-033/ISB |

**Supervised by**

Dr Haroon Ahmed Khan

Assistant Professor,

Department of Electrical and Computer Engineering

CU Islamabad

COMSATS UNIVERSITY ISLAMABAD

Feb 2021

***Final Approval***

*This Project Titled*

*Hand-Gesture based actuation using computer vision.*

*Submitted for the Degree of*

***Bachelor of Science in Electrical (Computer/Electronics) Engineering***

*by*

|  |  |
| --- | --- |
| **Name** | **Registration Number** |
| **Ali Arslan** | CIIT/FA17-ECE-004/ISB |
| **Usama Mustafa** | CIIT/FA17-ECE-042/ISB |
| **Raja Nofil Hassan** | CIIT/FA17-ECE-033/ISB |

*has been approved for*

COMSATS UNIVERSITY ISLAMABAD

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Supervisor*

*Name,*

*Designation*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Internal Examiner-1 Internal Examiner-2*

*Name, Name,*

*Designation Designation*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*External Examiner*

*Name,*

*Designation*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Head*

*Department of Electrical and Computer Engineering*

***Dedication***

In the name of Allah, the Beneficent and the Merciful, this thesis is dedicated to:

1. Our beloved Holy Prophet Muhammad (SAW) who has always been the light for our guidance and blessing. He is the brightest light for us in the darkest night, without his blessings nothing could have been accomplished.

***Acknowledgements***

We are very grateful to our Lord Allah Almighty for the blessings and opening up our minds, lifting us where we are today. We are nothing without the grace and mercy of our Lord Allah Almighty, ALHAMDULILLAH.

A special thanks to our supervisor Dr Haroon Ahmed Khan and Co-Supervisor Dr Syed Saud Naqvi for providing us GPU in their lab at VC-Technologies for this project. Their dedication towards teaching is priceless. Thanks for your help in every aspect and making it possible.

|  |
| --- |
|  |
| **Ali Arslan** |
| **Usama Mustafa** |
| **Raja Nofil Hassan** |

***Abstract***

With the increase in the applications of VR/AR, 3D hand gestures are increasingly being used as actuation input. Consequently, hand pose estimation is an active field of interest today. In this work we aim to contribute to optimization of existing 3D hand pose estimation models that use a depth camera. This work focuses on the algorithms and techniques to minimize latency for real time applications while improving accuracy. This would offer a low-cost neural network-based counterpart to existing systems which mainly use RGB camera (s).

Our project is “Hand-Gesture based actuation using computer vision”. The project uses Kinect depth sensor that is commercially available, deep learning and computer vision techniques. to develop an API which is cross platform cross compiler compatible achieving points to replicate hand gestures. The API provides points in 3D space which can be used for many real time scenarios such as imitation of gesture by a robotic arm or in a VR game, by integrating the API.

Beside this, The Project is a research and development based, Which does research optimization in terms of latency. We replicated (Ge L, 2019) research that uses 3D CNN, and introducing our modified networks the latency gets minimal. There are many approaches for the hand pose estimation such as using encoder decoder networks, 2D Convolution Neural Networks (CNNs), 3D Convolution Neural Networks (CNNs), Regression of latent heatmaps. The 3D CNN approach is best among all as the 3D CNN is more capable for the 3D spatial exploitation. We approached all the techniques stated above and will be explained further but the best technique we observed was 3D CNN proposed in (Ge L, 2019).

.

***Table of Contents***

[Chapter-1 *-* Introduction 1](#_Toc74319212)

[1.1 Background: 1](#_Toc74319213)

[1.2 Motivation: 1](#_Toc74319214)

[1.2.1 Hand-pose estimation: 1](#_Toc74319215)

[1.3 Scope: 2](#_Toc74319216)

[Chapter-2 - Literature Review 2](#_Toc74319217)

[2.1 Machine Learning: 2](#_Toc74319218)

[2.1.1 Supervised Learning: 2](#_Toc74319219)

[2.1.2 Unsupervised Learning: 3](#_Toc74319220)

[2.1.3 Reinforcement Learning: 3](#_Toc74319221)

[2.1.4 Deep Learning: 3](#_Toc74319222)

[2.1.5 Neural Networks: 3](#_Toc74319223)

[2.1.6 Dense Neural networks: 3](#_Toc74319224)

[2.1.7 Convolutional Neural Networks: 4](#_Toc74319225)

[2.1.8 2D CNNs: 4](#_Toc74319226)

[2.1.9 3D CNN 5](#_Toc74319227)

[2.1.10 Principal Component Analysis (PCA): 5](#_Toc74319228)

[2.1.11 Surrogate Function (Probability Model): 5](#_Toc74319229)

[2.1.12 Tree-structured Parzen Estimator ( TPE ): 6](#_Toc74319230)

[2.2 Early Developments: 6](#_Toc74319231)

[2.2.1 Existing 2D CNN methods: 6](#_Toc74319232)

[2.2.1.1 Encoder decoder Networks: 6](#_Toc74319233)

[2.2.2 Existing deep learning methods: 7](#_Toc74319234)

[2.2.3 Existing 3D CNN methods: 7](#_Toc74319235)

[Chapter-3 - Methodology 7](#_Toc74319236)

[3.1 Prediction Pipeline 7](#_Toc74319237)

[3.2 Neural Network 8](#_Toc74319238)

[*3.2.1 Neural Network Architecture* 8](#_Toc74319239)

[3.3 Input to the Neural Network 9](#_Toc74319240)

[3.3.1 3D Volumetric Representation 10](#_Toc74319241)

[3.3.1.1 Cartesian Representation of Depth Image 10](#_Toc74319242)

[3.3.1.2 Axis Aligned Bounding Box (AABB) 10](#_Toc74319243)

[3.3.1.3 Occupancy Grid 11](#_Toc74319244)

[3.3.1.4 Truncated Signed Distance Function (TSDF) 11](#_Toc74319245)

[3.4 Experimental Setup 12](#_Toc74319246)

[3.5 Testing and Benchmarking 12](#_Toc74319247)

[3.6 Real-Time Inference 14](#_Toc74319248)

[3.6.1 Python Prototype 15](#_Toc74319249)

[3.6.2 JIT Compilation with Numba 15](#_Toc74319250)

[3.6.3 C++ implementation 15](#_Toc74319251)

[3.6.4 Nvidia Cuda® Paralellization 16](#_Toc74319252)

[3.7 Project Execution Pipeline 19](#_Toc74319253)

[Chapter-4 - System Design 19](#_Toc74319254)

[4.1 Hardware: 19](#_Toc74319255)

[4.1.1 Kinect V2: 19](#_Toc74319256)

[4.1.1.1 RGB Camera: 20](#_Toc74319257)

[4.1.1.2 3-D Depth Sensor: 20](#_Toc74319258)

[4.1.1.3 Tilt adjustment: 20](#_Toc74319259)

[4.1.2 GPU 20](#_Toc74319260)

[4.2 Software: 20](#_Toc74319261)

[4.3 Future Discussion 21](#_Toc74319262)

[Bibliography 22](#_Toc74319263)

[References 22](#_Toc74319264)

***List of Acronyms***

CNN……………………………………………… Convolutional Neural Network

API………………………………………………... Application Programming Interface

KNN………………………………………………. K nearest neighbor

PCA………………………………………………. Principle component analysis

TSDF……………………………………………… Truncated signed distance function

CV………………………………………………… Computer vision

NLP………………………………………………. Natural Language Processing

HCI………………………………………………. Human computer interaction

CV………………………………………………… Computer vision

IR…………………………………………………. Infrared

RGB………………………………………………. Red, Green, Blue

***List of Figures/Tables***

[Figure 1:A figure showing connections, weights and bias in a neural network 4](#_Toc74321963)

[Figure 2: A dense neural network. 4](#_Toc74321964)

[Figure 3: A CNN architecture. 5](#_Toc74321965)

[Figure 4: 2D kernel convolving 5](#_Toc74321966)

[Figure 5: 3D kernel convolving 6](#_Toc74321967)

[Figure 6: Response for 2 hyperparameters 6](#_Toc74321968)

[Figure 7: Encoder-Decoder Network 7](#_Toc74321969)

[Figure 8: Pipeline for prediction of x,y,z coordinates of Hand Joints from TSDF 8](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321970)

[Figure 9: (Ge L, 2019) architecture. 9](#_Toc74321971)

[Figure 10: Our modified architecture 10](#_Toc74321972)

[Figure 11: Depth image 11](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321973)

[Figure 12: Point Cloud from Depth Image 11](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321974)

[Figure 13: AABB and voxel space for TSDF. 11](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321975)

[Figure 14: 3D occupancy grid. 12](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321976)

[Figure 15 Average Errors of each Model 14](#_Toc74321977)

[Figure 16 Time of Inference of Each Model. 14](#_Toc74321978)

[Figure 17 Mean error of joints 15](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321979)

[Figure 18: Difference between a 4-Core CPU and an Nvidia GPU 17](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321980)

[Figure 19: C++ Program Sequential Execution 18](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321981)

[Figure 20: A Cuda® Kernel to compute TSDF 19](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321982)

[Figure 21: Memory Initialization in Cuda® 19](file:///C:\Users\lenovo\Downloads\thesis_4v2.docx#_Toc74321983)

[Figure 22: Kinect V2. 20](#_Toc74321984)

[Figure 23: Kinect use cases. 20](#_Toc74321985)

[Figure 24: GPU specifications 21](#_Toc74321986)

[Figure 25: Comparison of openNI and Microsoft SDK 21](#_Toc74321987)

# Chapter-1 *-* Introduction

## 1.1 Background:

In recent years after the introduction of cheaper depth cameras such as Intel RealSense, Kinect. Challenges for pose estimation tending towards many solutions and real-time 3D hand pose estimation has instigated a lot of research attention in recent years (Huang, 2001). In 1994 the first efforts to tackle the problems were made (James M. Rehg, 1994). There are many applications such as action recognition, gesture recognition, and sign language interpretation. Till the challenge is to achieve accurate and robust hand pose estimation with unmarked visual inputs, variation in hand poses, similar fingers in the depth visual inputs. Much research has been done and is in progress to make a robust and effective hand pose estimation.

Many recent works focused on Convolution Neural Networks (CNN) (Ge L, 2019) for the 3D hand pose estimation and achieved good performance results as large datasets are readily available. Many researchers used these methods that feed the depth images to 2D CNNs that produce 3D joints (Rehg, 1994)). This method is not effective to achieve the real-time level of latency, as 2D CNNs are not sufficient for 3D hand pose estimation due to the lack of spatial information in three dimensions. The results from the 2D CNNs setup are inaccurate which can be made better by applying multi-view 2D CNNs which do not fully utilize the 3D spatial information from depth images. Improving the performance from multiple view 2D CNNs increases model complexity and latency. The 3D CNN-based hand pose estimation approach is much effective because 3D CNN fully utilizes the three-dimensional nature of depth images. The work in (Ge L, 2019) used 3D CNNs which capture the 3D spatial structure from the segmented data extracted from the depth images, from which a 3D point cloud of the hand is encoded as 3D volumes storing the Accurate Truncated Signed Distance Function (TSDF). The output is a reduced dimensional representation of 14 Hand Joints by applying PCA.

## 

## 1.2 Motivation:

As AR/VR-based applications are getting very common, key interest area for them gaming them is gaming industry, along with areas where human interaction is hazardous such as mechanical gloves and Expensive solutions. Till the mode of using hand directly for Human-Computer interaction (HCI) was glove based techniques, after the evolution of cheaper depth sensors and large datasets the hand-pose estimation is evolving day by day having better performance, so it is a great area of interest for researchers to eliminate expensive glove-based techniques and provide a robust solution for HCI using CV.

### 1.2.1 Hand-pose estimation:

Hand-pose estimation is endeavored to detect joint locations on a human hand using input images or a video stream. There are many applications related to hand pose estimation:

* + Sign Language Interpretation
  + HCI for modern cars
  + Remote multi-joint actuators activation
  + VR gaming

As there are some tasks and places where direct human contact can be hazardous and there is a need for something to prevent damage, the hand-pose estimation can eliminate the problem. As recently due to a global pandemic, where touching an infected person is harmful so by using pose estimation and computer vision robotic arm can be controlled using hand gestures.

## 1.3 Scope:

The project has been designed to develop a cross compiler cross-platform compatible API that a user can use in any type of application to replicate hand gestures by the points given by our API. There are vast applications of hand-pose estimation the API we provide produces detected points that can be used in any application either in VR games, HCI, or remote joint actuation. The main goal is to provide the end-user a seamless control over hand gestures with very low latency over any application. This can result in an API development for use in third party software and be really useful with commercial audience. The API can be cross platform as all the libraries used in this C++ code are cross platform compatible.

This works aims to replicate the research paper “ **3D Convolutional Neural Networks for Efficient and Robust Hand Pose Estimation from Single Depth Images**” published in IEE CS by (Liuhao Ge1, Hui Liang). The research proposed our new modified network having less latency being more optimal than in the paper.

# Chapter-2 - Literature Review

## 2.1 Machine Learning:

Machine learning by the definition of Arthur Samuel is the concept and study of computer systems that make the computer systems handle different tasks without being programmed explicitly. The study of machine learning involves machine learning algorithms it gets the ability to learn and make decisions using data. There are three major types of machine learning:

### 2.1.1 Supervised Learning:

Supervised machine learning involves labeled data on which the supervised machine learning algorithm/function learns and trains and gives predictive outputs by mapping the inputs to predictive outputs. It involves training data based on which the model gives predicted output. Some basic supervised learning algorithms are:

* Linear regression
* Logistic regression
* Naïve Bayes

### 2.1.2 Unsupervised Learning:

Unsupervised learning is a machine learning technique that does not require a labeled data either the algorithm searches similar patterns in provided data and forms its groups and then by these patterns predict outputs. Some of the unsupervised learning algorithms are:

* KNN
* K-means clustering
* PCA

### 2.1.3 Reinforcement Learning:

Reinforcement learning is a machine learning technique that does not requires data to be fed either it makes decisions and sequences from its own by observing the environment and adapts suitable techniques to reach the goal.

### 2.1.4 Deep Learning:

Deep learning is a subfield of machine learning concerned with the algorithms that perform complex tasks based on data-driven techniques. The accuracy of deep learning approaches is highly dependent upon the feature engineering, quality, and amount of data. Deep learning involves neural networks inspired by the human brain. Deep learning outperforms in the scenarios of complex tasks such as Computer Vision, Natural Language Processing, etc. The model here is deep with many layers and these deep models are provided large data. Deep learning is now in our everywhere either in autonomous vehicles, Defense systems, software applications, mobile apps, and much more.

### 2.1.5 Neural Networks:

Neural networks are the collection of nodes connected to each other imitating human brain neurons, each node represents a function, takes inputs, and gives outputs that go to another neuron. Neural networks are a core part of deep learning. Neural networks are useful in many applications such as clustering, classification, regression, and time-series predictions. In neural networks, the input is broken down and fed into the different layers of abstraction

Diagram

Description automatically generated

Figure :A figure showing connections, weights and bias in a neural network

### 2.1.6 Dense Neural networks:

Dense neural networks are networks where all the nodes have inputs from all the nodes from the previous layer. These are fully connected layers. The dense networks extract all the possible learning features from the previous layers. The nodes have linear functions as all the nodes in a layer are connected to all the nodes of the previous layer.

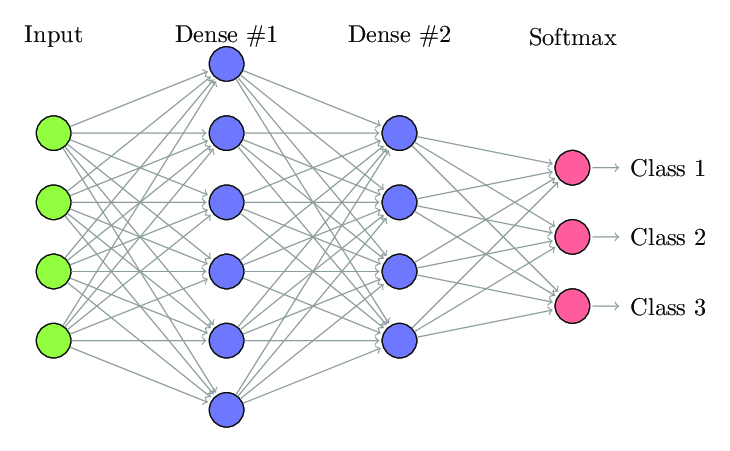


Figure : A dense neural network.

### 2.1.7 Convolutional Neural Networks:

A CNN is a deep learning-based algorithm mainly used for visual inputs. It is similar to an ordinary neural network. These networks involve convolution layers that compute the outputs by the dot products of weights and other connections. CNNs are best for exploiting visual data. In the CNNs there is a filter that rotates above the imagery input data and convolves with the data to produce arrays that further fed into the network. Each node is fed a pixel value that defines a node's weight. The kernel in the CNN makes it robust to extract spatial information from imagery data.

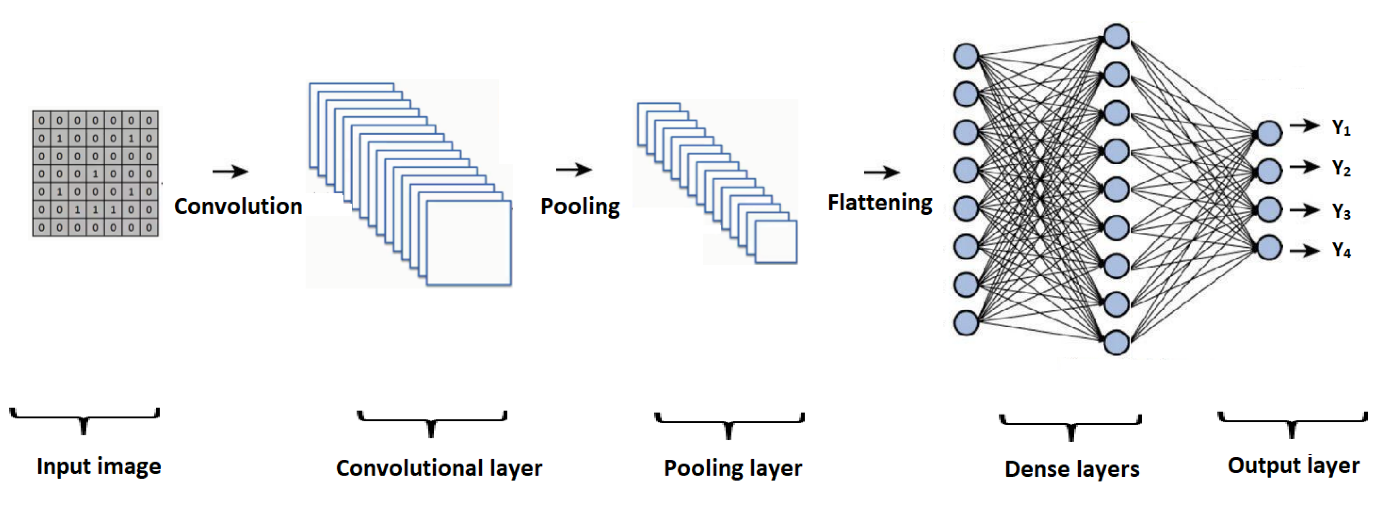


Figure : A CNN architecture.

In the CNN architecture, the input data is first convolved with the filter forms an array that is fed into the network further.

### 2.1.8 2D CNNs:

2D CNN involves a 2- dimensional kernel/filter that rotates and maps in the input data the kernel moves in 2 directions the input data is in the form of a slice. This approach is majorly used in CNN with images at the input. The kernel moves in the 2D plane hence exploiting the boundaries and other complex features in the image.

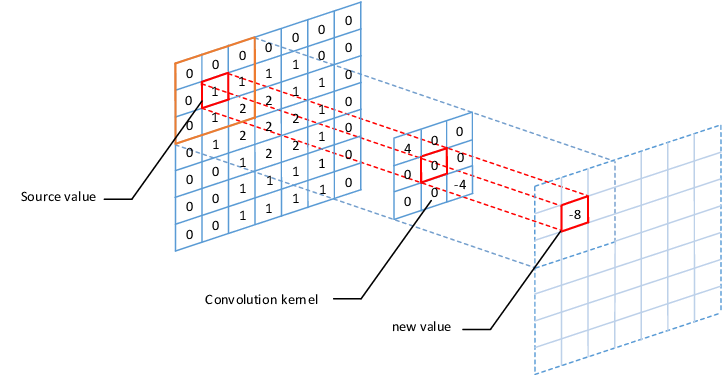


Figure : 2D kernel convolving

### **2.1.9 3D CNN**

3D CNN involves a 3- dimensional kernel that rotates in the 3 directions and maps to input data to provide output arrays. 3D Convolution kernels are applied when the input consists of volumes rather than 2D planes. 3D Convolution can detect features along with the depth of the volume. It is being used in video action recognition. In video action recognition the input is a video consisting of multiple frames batched together in the 3rd axis making a volumetric input.

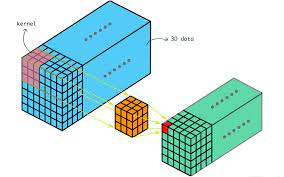


Figure : 3D kernel convolving

### 2.1.10 Principal Component Analysis (PCA):

PCA is a dimensionality reduction technique used to reduce the dimensions of feature state space. PCA allows recognizing patterns and correlations in a dataset by which it can be transformed into a lower dimension without the loss of important information. As high dimension data is sometimes become complex to process because of non-linear boundaries and disparity in the features that result in an increase of computational time.

### 2.1.11 Surrogate Function (Probability Model):

A surrogate function is an optimization function used to substitute expensive and latent models to simulate with the same parameters, it is a pre-trained model that plays a role in the efficiency and optimization of a model.

The surrogate function is the response function or response surface it tends to exploit the search space for the best loss or accuracy. It takes into account previous results and uses probabilistic calculations in order to find future best hyperparameters. The main objective of the surrogate function is to reduce the number the times the objective function needs to run. There are many forms of surrogate function including Bayesian optimizations. Figure 2.2 shows the response surface for only two hyperparameters:

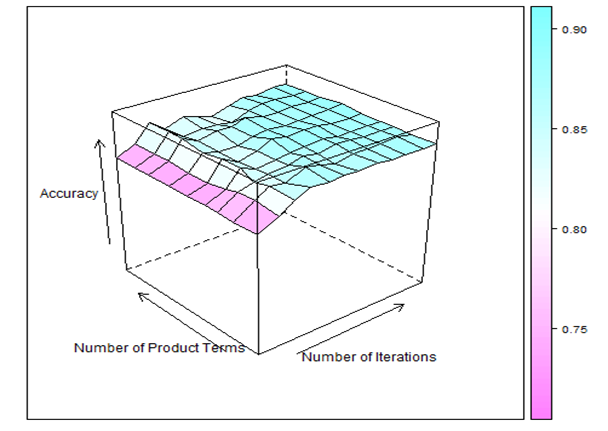


Figure 6: Response for 2 hyperparameters

### 2.1.12 Tree-structured Parzen Estimator ( TPE ):

The Tree-Structured Parzen Estimator (TPE) algorithm is designed to add quantization hyperparameters to detect quantization configurations that meet the expected accuracy and provide the best latency improvements. TPE is an iterative process using a history of tested hyperparameters to create a possible model.

TPE method models P (x | y) and P (y) where x represents hyperparameters and y corresponding quality points. P (x | y) is measured by modifying the production process of hyperparameters, including pre-parametric configuration distribution.

## 2.2 Early Developments:

## 2.2.1 Existing 2D CNN methods:

Regression of 3D hand features using a 2D convolutional neural network is a very active area of research. The work in (Jonathan Tompson, 2014) uses an encoder-decoder network which estimates 2D location using direct and latent heat maps and 3D locations using vector representations. Images of hands are fed to the encoder-decoder network, which produces heat maps and features separately, which are then concatenated.

### 2.2.1.1 Encoder decoder Networks:

Encoder decoder networks are a strategical development of Recurrent neural networks, it's an old machine learning technique having end-to-end sequence predictions. The network involves two recurrent neural networks one part is an encoder and the other is a decoder. The encoder network encodes the input sequel and then after the last layer at the end the decoder network decodes the encoded input features process them to make output predictions.

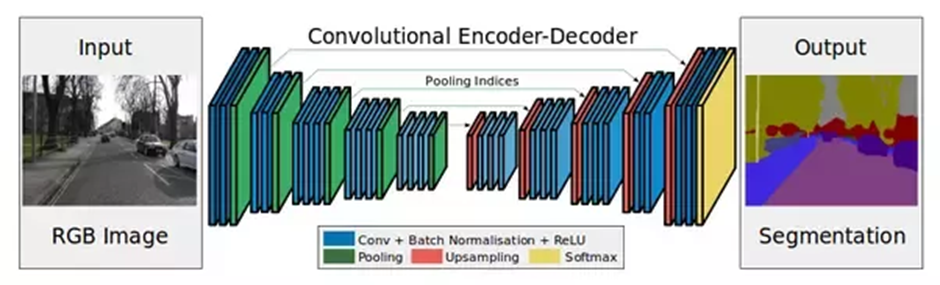


Figure 7: Encoder-Decoder Network

This approach was used by Ukraine catholic University for the 3D hand-pose estimation.

In this approach, the encoder-decoder network is fed with the input imagery. Which the network produces 2D heat maps that are then concatenated to produce 2D and 3D vector coordinates, these coordinates are then exploited in 2D and 3D space. The dataset used is synthetic data and the real datasets had only 5 key points so real-time implementation is not possible.

In the encoder-decoder networks, the number of neurons are large due to which the model becomes complex with many layers and neurons starts to increase complexity. The network is too much latent. It cannot be used in real-time.

## 2.2.2 Existing deep learning methods:

More recently, the AlexNet (Markus Oberweger, 2020) showed much better results, and deep learning is more and more involved as it produces accurate results and is simpler to implement. But for real-time consideration, it does not work due to its huge latency and demanding to compute capability.

## 2.2.3 Existing 3D CNN methods:

For real-time applications, 3D convolutional neural networks look more promising (Ge L, 2019). Due to the nature of 3D CNNs to better represent 3D features, the network can be much simpler and smaller, thus resulting in reduced latency. In the preprocessing the depth images were converted to volumetric representation i.e. TSDF.

This ground truth for the input TSDF was the lower dimensional embedding of absolute hand joints, so the degree of freedom could be increased. In this approach, the author has proved that it is better to predict lower embedding rather than absolute points.

# Chapter-3 - Methodology

## 3.1 Prediction Pipeline

There are two Neural Networks in the pipeline. One Neural Network is producing lower-dimensional embedding from the TSDF. The second Neural Network is producing 42 points from the lower 30-dimensional embedding. The original shallow network and our modified 3D CNN model, both take a volume input of voxels containing accurate TSDF. The network output is a tensor containing a lower-dimensional embedding of 30 features that are passed through a neural network to transform these lower dimension embeddings to 42, 3D joint location. It is better to predict the 3D spaces by using the lower dimension embedding rather than absolute spaces (Guo, 2017). The lower-dimensional output from the neural network is passed to a densely connected layer to produced 42 points. These 42 points are absolute 3D locations of the 14 hand joints. Each joint is represented by three coordinates X, Y, Z leading to 42 points. The flow of the input leading to absolute X, Y, Z coordinates of 14 points could be observed below:

Diagram

Description automatically generated with medium confidence

Figure 8: Pipeline for prediction of x,y,z coordinates of Hand Joints from TSDF

## 3.2 Neural Network

Artificial neural networks, usually simply called neural networks, are computing systems vaguely inspired by the biological neural networks that constitute animal brains. An Artificial Neural Network is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain.

### **3.2.1 Neural Network Architecture**

In this research-based project, various neural networks have been benchmarked based on latency and accuracy. The Neural network designed by us performs better in terms of both accuracy and latency. The first neural network that was implemented was a shallow plain network (Ge L, 2019). The architecture of the neural network can be seen in the figure below:

Diagram

Description automatically generated

Figure 9: (Ge L, 2019) architecture.

This neural network was modified in such a way that the number of filters at each layer were kept constant so latency would remain the same. The modified network could be seen in the figure below:

Diagram

Description automatically generated

Figure 10: Our modified architecture

The modification was made such that the diversity of filters at each layer was increased. Each layer now was made of both 5x5 and 3x3 filters. This approach allowed different features to be captured by the neural network rather than in the original network proposed by (Ge L, 2019) only one type of filter has been used at a single stage. Two neural networks were trained for producing a 3D representation of 14 hand-joints. The First Neural network has 1024 neurons in the first layer and 42 in the last layer. The second Neural Network has 2816 neurons in the first layer, 2304 neurons in the second layer, and 42 neurons in the final layer. The number of neurons at each layer was calculated by using the surrogate function to model the architecture of the neural network. The surrogate function used was Tree-Structured Parzen Estimator.

## 3.3 Input to the Neural Network

Previous work on hand pose estimation relied heavily on encoder-decoder methods to predict the 3D hand points. This approach has a very glaring flaw as it relies on 2D data to extrapolate 3D information. This work approaches this problem from another angle. Feature engineering is being used to modify a 2D depth image into a 3D volume with voxel resolution being set at compile time.

### 3.3.1 3D Volumetric Representation

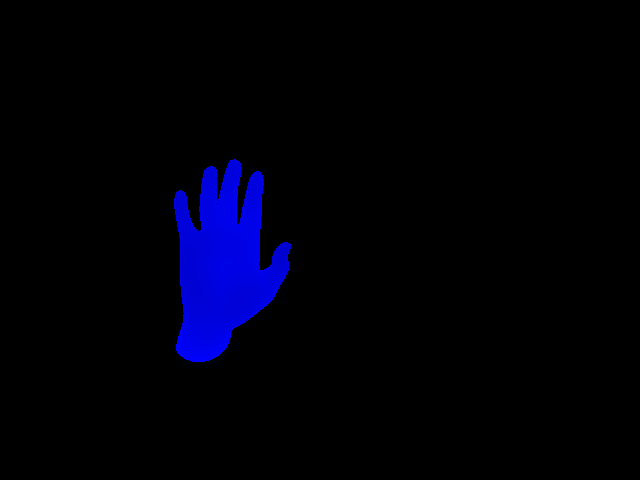
This approach starts from a depth image from a Microsoft Kinect® depth module. This depth is then converted into a 3D point cloud that just takes the depth data and interprets it as the Z dimension being placed based on the depth at that X, Y coordinate. This point cloud is then used to define an Axis Aligned Bounding Box (AABB) for these 3D points. This AABB is then used to create an occupancy grid of our desired volume resolution value. This gives a starting point to carve the voxels values based on the point cloud. The initial values of this voxel grid do not matter and will be changed in the future. The next step is to create a Truncated Signed Distance Function (TSDF) which can be seen as the actual hand surface. But the surface is not to be taken accurately as it will always be an estimation without any regard to the quality of the TSDF computation. This limitation derives from the fact that the depth camera only observes the hand from one axis. However, the use of a 3D mesh (synthetic or photogrammetry) can result in a much accurate representation of the human hand.

Figure 11: Depth image

### 3.3.1.1 Cartesian Representation of Depth Image

Figure 12: Point Cloud from Depth Image

The depth image does have 3D information but from one field of view. This depth of a pixel can be interpreted as an extra dimension for the point cloud. This dimension included describes each pixel with discrete point values in space. The figure included demonstrates this depth description with a reasonable degree of accuracy. The color of the visualization is based on the value of the Z-axis (depth) with blue being close to the camera and red being further away. Note that there are no other points besides the hand points. This gives the neural network a really clean image without unnecessary background information. Segmenting the depth image is therefore absolutely paramount as the entirety of the pipeline from this point forwards depends on the point cloud being as clean as possible.

### 3.3.1.2 Axis Aligned Bounding Box (AABB)

Figure 13: AABB and voxel space for TSDF.

AABBs are extensively used as a primitive in 3D graphics rendering. AABBs have only two attributes both being in 3D space. The First defines a minimum bound and the second defines the maximum bound. Only having two attributes for a point cloud bound can make AABBs really computation efficient. Traditionally this primitive is used for object intersection to find if an object is behind another object or overlapping if at all, but this work is not using this primitive in that manner. This geometry is instead used to define an outline for the voxel space for our TSDF. The figure shows the AABB as well as the voxel grid outline. The voxel grid will always be a cube to avoid dynamic voxel values on each frame. Since the outline is static and fixed, it is being computed in the program initialization phase and is not being computed for each frame. Instead, the points are being normalized between the min and the max bounds of AABB in the point cloud phase.

### 3.3.1.3 Occupancy Grid

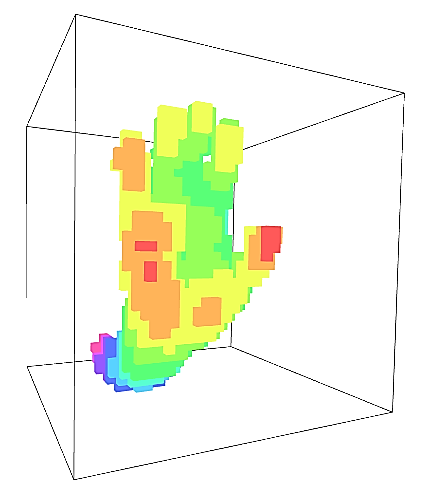
Occupancy Grid is a binary grid in 3D space, representing the voxels that have a point in it as ones and the voxels that are empty as zeros. For this work, a volume resolution value of 32 is being used. This implies that each volume has 32x32x32 voxels. This can be higher but it is observed to be highly computationally expensive with no discernable accuracy improvement. The increase in computation cost is due to the fact that the algorithm scales with a big O notation of O(n3) where n being the volume resolution value.

Figure 14: 3D occupancy grid.

The algorithm being used in this work to determine whether or not a voxel contains a point, can be computed using Chebyshev Distance (Ge L. 2019). Chebyshev distance is also called maximum value distance. It examines the absolute magnitude of the differences between coordinates of a pair of points.

The above equation describes the algorithm to compute the occupancy grid of the voxel space. The Chebyshev Distance can be found as

This can be used to compute the occupancy grid for the whole voxel space. Now, the depth data is no more 2D but instead, it can be interpreted as 2.5D. Previously it is mentioned that a mesh has better 3D information than a depth image as it is truly 3D. For the 3D mesh, the preprocessing can be stopped at this point. This approach is feasible in a simulation environment or synthetic recreations but in the real world, the use of photogrammetry to construct a 3D mesh for each frame is not practical. Therefore, a depth image is more convenient. An occupancy grid of a hand contains no information about the signed direction of the 3D space, i.e. which voxels are in front of the hand or behind the hand. This is where the process is transferred to the next stage to interpolate the data.

### 3.3.1.4 Truncated Signed Distance Function (TSDF)

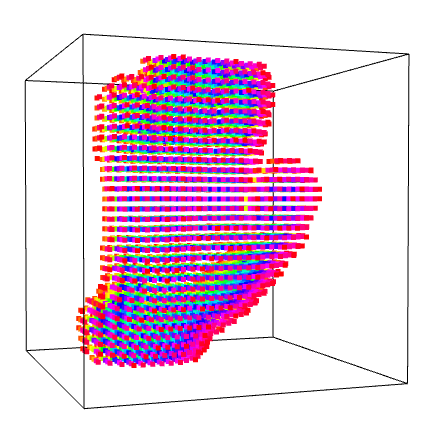
A Signed Distance Field (SDF) of a 3D space stores a distance value in each voxel and is found out by calculating the distance from the voxel center and each point in the point cloud and storing the minimum distance value in the voxel. The voxels in front of the boundary of the point have different signs than the voxels that are inside the point cloud. This value increases as the voxels move further away from the point. The value increases indefinitely without bounds if the voxels are too far away. This can be mitigated by applying a truncation value condition on the signed distance function and that’s where the name of the truncated signed distance function comes from. The truncation value determines how big the values have to be, to be assigned as unity, i.e. SDF value of a voxel after a certain threshold can be set as a constant, in this case, unity due to the normalization of the SDF values between 0 and 1.

Figure 15: A TSDF representation.

This stage is the most computationally expensive in our whole pipeline as it is required to calculate the distance between each voxel and each point in the point cloud to compute an accurate enough TSDF. Luckily this is highly parallelizable and this property allows the whole hand pose estimation to be run at near real-time speed. The different techniques used in this work to lessen the computation time for this algorithm are mentioned later on.

The actual algorithm implemented in code is quite simple but the fact that it has to be computed for each voxel and in each voxel it has to be computed for each point.

The equation here defines the behavior under which the algorithm is written. is the truncation distance and it defines how far away a voxel has to be from the closest point before it is set to 1 or -1 depending on the direction of the voxel. This can be varied but for this work, it is set as three times the length of each voxel. The figure shows the TSDF of a depth frame. The color values here represent the TSDF values of each voxel. The red voxels are ones and the blue voxels are zeros. They can be seen as the outer and inner surface of a hand respectively. This estimation is required because the full view of the hand from all angles cannot be provided in real-time with one camera. Now, this representation resembles very much the actual 3D mesh of the hand and thus can be taken as such and the 3D CNN can be trained on this data.

The work done by (Ge L, 2019) also mentions “Directional-TSDF (DTSDF)” which is calculated along the line of sight of a camera position defined manually, and thus the distance between all the points and all the voxel centers is not computed. Only the distance for voxels along the line of sight of the camera is being computed. This speeds up the process by a lot but it does have its limitations. This is not an accurate representation of it and only an approximation of the actual TSDF. Their work took three different DTSDF volume sets where, in each volume, the distance along one of the cartesian coordinates is computed, i.e. along X, Y, and Z axis. Therefore, three different volume sets must be fed to the 3D CNN to perform a prediction. This work opted to not go that route due to the simple reason that TSDF is a lot more accurate than DTSDF.

## 3.4 Experimental Setup

The replicated the implementation pipeline of Liuhao Ge (Ge L, 2019). The publicly available hand pose dataset (Tompson, 2014)is used in this work. The depth images from the NYU dataset were converted to accurate Truncated Signed Distance Function (TSDF) based volumes within axis-aligned bounding boxes (Ge L, 2019). Neural Networks with 3D convolution filters were used to train on these volumes. 14 Hand joints were selected for regression from the Neural Network (Ge L, 2019). The points representing 14 Hand Joints were normalized between zero and unity. The 42 3D features representing 14 Hand joint locations were reduced to 30 feature vector space using PCA as the most optimal number of features described in (Ge L, 2019). The shallow plain network in (Ge L, 2019) was implemented and modified in a way to produce better results. SGD optimizer performed best on all the models. All the models were trained with a learning rate of 0.01 initially, and then it was divided by 10 after every 5 epochs. All models were trained for 15 epochs with 3500 steps per epoch and a batch size of 16. The loss function was Mean Absolute Error. 10 percent of the data was used as a validation split. The lower Feature Representation of the 14 joints as the output of the neural network used fully connected layer(s) to reconstruct the 3D joint features.

## 3.5 Testing and Benchmarking

The test data from the NYU dataset was used for testing and benchmarking. The number of samples was 8252. The models were benchmarked based on both latency and accuracy. The accuracy was measured on the 3D representation of 14 hand joints rather than a 30-dimensional vector from the CNN. Two networks were trained for the conversion of 30- dimensional vector into the 3D points for 14 joints. Benchmark for both has been calculated individually. The other networks that were trained and benchmarked were MC3 (Tran, 2018), R(2+1)d-18 (Tran, 2018) and ResNet3D-18 (Tran, 2018).

**)/14**

**Average Errors of Each Model (%):**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **2 Layers** |  | **3 Layers** |
| **Ge-Le** | 10.13 | 9.33 |  |
| **Our** | 1.004 | 1.000 |  |
| **mc3\_18** | 1.74 | 1.74 |  |
| **r2plus1d\_18** | 1.54 | 1.54 |  |
| **resnet3d\_18** | 1.45 | 1.45 |  |

*\*Lower is better.*

**Table 1.1:** Average Errors of Each Model (%)

**Time of Inference of Each Model (ms):**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **2 Layers** |  | **3 Layers** |
| **Ge-Le** | 14.02 | 15.37 |  |
| **Our** | 29.54 | 30.76 |  |
| **mc3\_18** | 26.37 | 25.62 |  |
| **r2plus1d\_18** | 32.25 | 32.00 |  |
| **resnet3d\_18** | 29.77 | 29.88 |  |

*\*Lower is better.*

## 

## 

Figure Mean error of joints

## 3.6 Real-Time Inference

The excellent results of benchmarking and testing allow this work to move forward to the end product development phase. There are still some challenges that need to be overcome before this prototype turns into an end product. Most notable of which is reducing the preprocessing of input data to the 3D CNN. The 3D CNN in combination with the Inverse PCA model does not cost that much computation resources. The majority of time to predict one instance is taken up by the computation of TSDF for 3D CNN. Three different approaches were developed to reduce this time to near real-time.

### 3.6.1 Python Prototype

The initial prototype was completely developed in python. Python is a very flexible work environment and requires very little work to modify the flow of the pipeline. This is the reason, this language was chosen for prototyping and developing, and visualizing the different stages of the pipeline. The computation time for Accurate TSDF on python was somewhere between 10 seconds to 15 seconds. The majority of code for this computation was written in Numpy data formats. Numpy’s backend is written in C so it resulted in a significant improvement over pure python implementation. From this point forwards all of our implementation in python is utilizing the excellent Numpy backend for computation and visualization. Even with the speedup that Numpy provides our computation times are still nowhere near the real-time requirements.

### 3.6.2 JIT Compilation with Numba

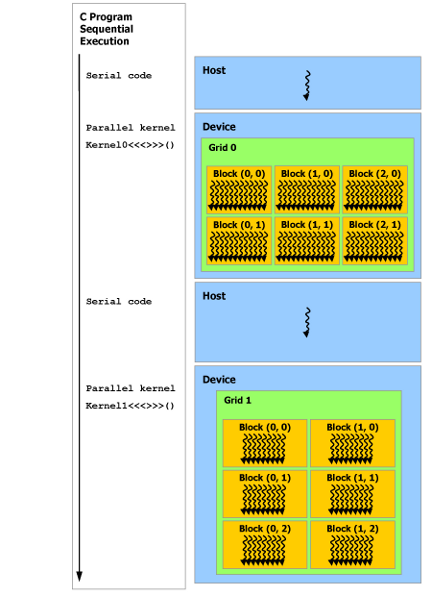
To meet the real-time requirements while still keeping the codebase in python, Numba was used. Numba is a prerelease module for python which allows its users to compile python code in runtime. Python is an interpreted language and thus it is not compiled in assembly but instead, is reduced to byte code. This byte code has more advanced features than what the raw assembly of an x86 architecture can provide. This results in a relatively slow code that is convenient but not very computationally efficient. Numba can solve this issue somewhat by further reducing that byte code to actual x86 assembly. This is done by compiling the python code in runtime. The compiler used by Numba is LLVM. Note that the whole TSDF implementation is only compiled on the first iteration or frame and all the subsequent frames run on that compiled binaries. This Numba implementation still required some python code modifications because Numba is still in prerelease and does not support all the python classes and methods. The TSDF computation time fell to around 7 seconds. The time was expected to be lower but this time makes sense because the majority of our codebase was already in Numpy and the speedup obtained with Numba was negligible. This is still not enough for the real-time implementation of this work.

### 3.6.3 C++ implementation

The TSDF computation time, still nowhere near run-time, forced the us to rewrite the whole codebase in C++. This language compiles directly to the x86 assembly with the target platform being configurable. This means the code can be compiled for both 32-bit and 64-bit architectures. TSDF computation in C++ was first written without multicore support. The entire computation was being done on one CPU core. This is not very efficient and does not take full advantage of the CPU multicore computation.

The TSDF computation time for Debug mode without any compiler-assisted code optimization was 5 to 6 seconds. Debug mode allows its users to easily access the data in RAM at run-time to find out any exceptions that need to be handled.

A modern Intel® processor has a type of instructions called Advanced Vector Extensions 2 (AVX2). These instructions are essentially SIMD instructions and are not used in Debug mode. To take advantage of these instructions the code should be compiled in Release mode. In Release mode, no variables or classes can be accessed at run-time because the compiler makes fundamental structural changes to the program code. For example, if a variable is initialized but is used as a static variable, the Debug mode does not change that, and it can be accessed at run-time. In Release mode, however, this is not the case because the compiler essentially deletes that variable to not require frequent memory calls to the RAM. The use of this example variable is not exactly a bad code implementation because it can make a code more readable.

The time with Release mode fell to around 1 to 2.5 seconds. The compiler settings in Release mode were, set to favor speed instead of code size (O2, Ot). The frame pointer was not omitted for stack allocation as the program calls functions that had relatively large sizes. Cross module optimization was also enabled because the codebase spaned several translation units.

### 3.6.4 Nvidia Cuda® Paralellization

Due to the highly parallel nature of our TSDF computation, it was theorized that it can take advantage of parallel computation of a GPU. The code can calculate TSDF for each voxel independently and concurrently at the same time instead of serially like on a CPU. Out of all the GPU architectures, Nvidia Cuda® was chosen due to its relatively advanced architecture and excellent developer support from Nivida. Cuda® compiles to ptx instructions which run directly on GPU. The data, however, does has to be initialized on the CPU and then transferred to the GPU. This transfer speed is strictly dependent on the PCI-Express interface between CPU and GPU. The development computer had a PCI-E Gen 3.0 interface. Further speedup can be expected if the code is ported to a Gen 4.0 or Gen 5.0 interface.

Cuda® has a very different coding paradigm than CPU coding practices. Nvidia calls its programming approach Single Instruction Multiple Threads (SIMT). Similar to SIMD instructions which are bound to only one instruction, a SIMT approach can include a whole function level code that can have its own memory and can also interact with a global memory. Nvidia splits its GPU resources into three classes. Grids, Blocks, and Threads. Nvidia has a core type called Cuda Core. Which Nvidia markets as the parallel performance ability of a GPU. Nvidia also calls this Cuda core a streaming processor (SP). Eight of these streaming processors make up a streaming multiprocessor (SM). One instruction is executed on an SM instead of an SP. This means all SPs inside an SM execute the same ptx instruction. Nvidia also creates another execution class called warp. Inside a warp, 8 SPs take 4 clock cycles to execute a single instruction on multiple threads. Each warp contains 32 threads.

In C++ Cuda® programming, each thread is assigned a block address as well as a Grid address. Each thread is inside a block and each block is inside a grid. All grids execute concurrently and each thread can be considered as an instance of a for loop in conventional CPU programming. Except that this is being done in parallel. The indexing of this for loop analogy in Cuda® is determined through the address of a thread by also taking into account the address of the block and grid.

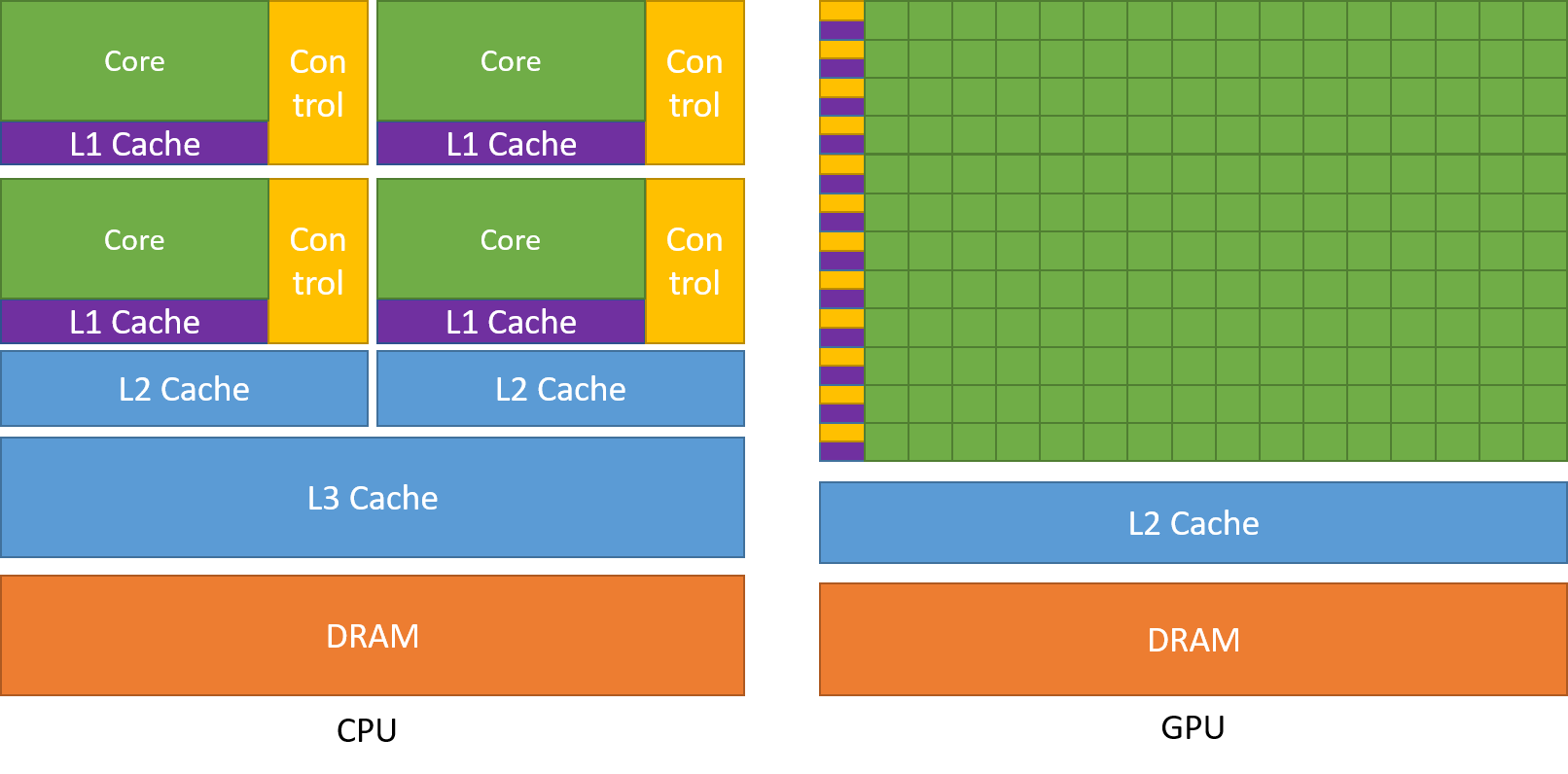
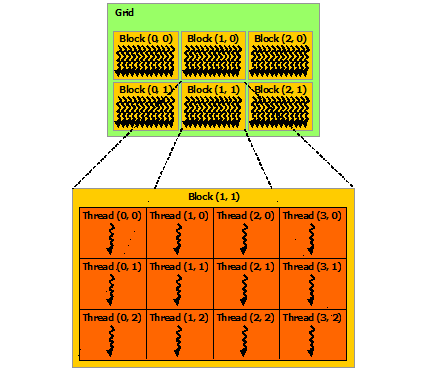
A C++ program is executed sequentially. Once it reaches a Cuda® kernel call, it transfers the instruction pointer to the GPU and the GPU takes the control. Note that all data should be transferred to the GPU explicitly to avoid memory access errors. The GPU cannot access CPU memory and vice versa. Nvidia differentiates between these two entities with a device and host nomenclature. A device is a Cuda capable GPU and a host is a CPU.

Figure 18: Difference between a 4-Core CPU and an Nvidia GPU

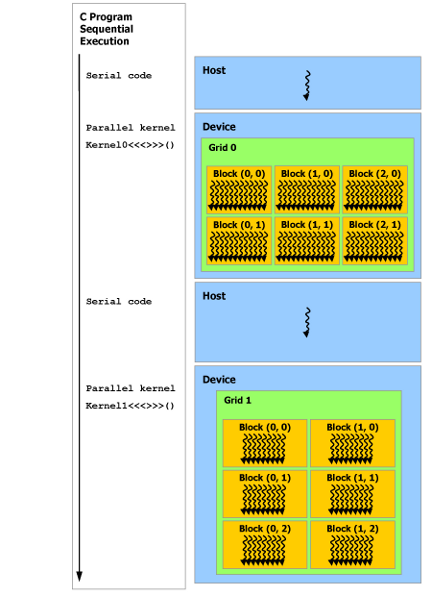
TSDF compute on GPU was extremely important to be able to run this work in real-time. Since this work focuses on Accurate TSDF, the distance between all points and voxel centers still needs to be calculated for each voxel. All the voxels are being calculated independently but all the points distances for each voxel are not. This creates a bottleneck and dependency in our pipeline and the program is now highly dependent on the clock speed of the GPU. Since the program scales really well with clock speed instead of Cuda Cores, further speedup can be expected by executing this work on newer Nvidia RTX 30 series GPUs.

Figure 19: C++ Program Sequential Execution

An average time of 250 ms was observed for each frame to compute a TSDF and infer on the Neural Network. Note that the Neural Network was also on GPU.

Figure 20: A Cuda® Kernel to compute TSDF

Figure 21: Memory Initialization in Cuda®

\_\_global\_\_ void tsdf\_compute\_kernel(double\* d\_tsdf\_data, int\* d\_x, int\* d\_y, int\* d\_z,

 double\* d\_volume\_attributes)

{

    const int x = blockIdx.x \* blockDim.x + threadIdx.x;

    const int y = blockIdx.y \* blockDim.y + threadIdx.y;

    const int z = blockIdx.z \* blockDim.z + threadIdx.z;

    if (x >= 32 || y >= 32 || z >= 32)

        return;

    double voxel\_center\_x = (d\_volume\_attributes[0] + x \* d\_volume\_attributes[3]) + \

d\_volume\_attributes[3] / 2;

    double voxel\_center\_y = (d\_volume\_attributes[1] + y \* d\_volume\_attributes[3]) + \

d\_volume\_attributes[3] / 2;

    double voxel\_center\_z = (d\_volume\_attributes[2] + z \* d\_volume\_attributes[3]) + \

d\_volume\_attributes[3] / 2;

    int distance\_index = 0;

    double distance\_min = 1.73205080757 \* d\_volume\_attributes[3] \* 32;

    for (int l = 0; l < (int)d\_volume\_attributes[5]; l++)

    {

        double a = voxel\_center\_x - (double)d\_x[l];

        double b = voxel\_center\_y - (double)d\_y[l];

        double c = voxel\_center\_z - (double)d\_z[l];

        double dist = sqrt(pow(a, 2) + pow(b, 2) + pow(c, 2));

    }

    distance\_min = distance\_min / d\_volume\_attributes[4];

    d\_tsdf\_data[x + 32 \* (y + 32 \* z)] = distance\_min;

}

cudaMalloc((void\*\*)&device\_x, pcd.number\_of\_points \* sizeof(int));

cudaMalloc((void\*\*)&device\_y, pcd.number\_of\_points \* sizeof(int));

cudaMalloc((void\*\*)&device\_z, pcd.number\_of\_points \* sizeof(int));

cudaMalloc((void\*\*)&device\_volume\_attributes, 6 \* sizeof(double));

cudaMalloc((void\*\*)&device\_tsdf\_data, 32 \* 32 \* 32 \* sizeof(double));

cudaMemcpy(device\_x, pcd.points\_x, pcd.number\_of\_points \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(device\_y, pcd.points\_y, pcd.number\_of\_points \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(device\_z, pcd.points\_z, pcd.number\_of\_points \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(device\_volume\_attributes, host\_volume\_attributes, 6 \* sizeof(double), \

cudaMemcpyHostToDevice);

# Chapter-4 - System Design

## 4.1 Hardware:

The hardware used in the project is a 3D sensor and a laptop having a discrete GPU by Nvidia®. The KinectV2 is used for the collection of real-time input imagery data which the laptop processes to get desired results. the 3D sensor which we have used is commercially available Microsoft® KinectV2.

### 4.1.1 Kinect V2:

KinectV2 is a 3D depth-sensing module launched by Microsoft®. It consists of an RGB camera, IR camera, and an IR projector/Time of flight sensor.

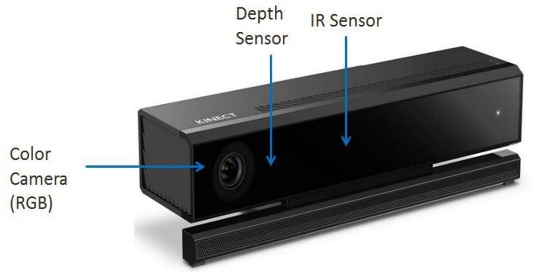


Figure 22: Kinect V2.

It is an RGB-D sensor that provides a synchronized color and depth images. It was initially used as an input device by Microsoft for the Xbox game console. With a 3-D human motion capturing algorithm, it enables interactions between users and a game without the need to touch a controller. Recently, it became famous among computer vision research as the computer vision research community has found that Kinect's advanced depth-sensing technology can be expanded far beyond play and at a much lower cost than traditional 3-D cameras.

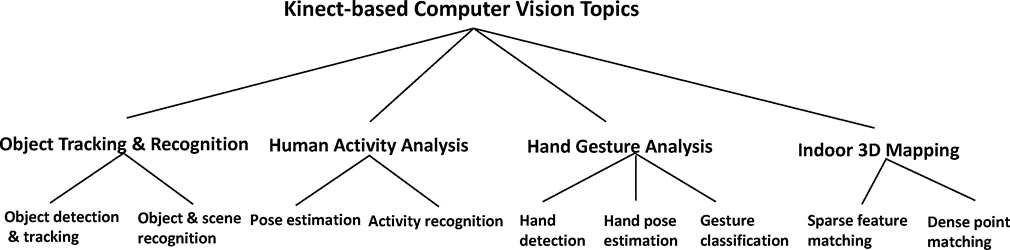


Figure 23: Kinect use cases.

The Kinect V2 has a depth resolution of 512 x 424 pixels having a field of view (FoV) providing an average of around 7 x 7 depth pixels per degree. As a result of using time-of-flight as the core mechanism for depth capture, each pixel in the depth image contains a real measured depth value (z-coordinate). Each component of Kinect hardware is described below:

### 4.1.1.1 RGB Camera:

Provides three basic color components of the video. The camera operates at 30 Hz and can

provide images with 640 × 480 pixels with 8 bits per channel. Kinect also has the option to produce high-resolution photos, which work in 10 frames at the resolution of 1280 × 1024 pixels.

### 4.1.1.2 3-D Depth Sensor:

Has an IR laser projector and IR camera. Together, the projector and the camera create a depth map, which provides details of the distance between the object and the camera. The sensor has a working range of 0.8m – 3.5m, and output video with a frame rate of 30 frames per second 640 × 480-pixel resolution. The angular field for the view is 57◦ horizontally and 43◦ vertically.

### 4.1.1.3 Tilt adjustment:

It is the basis of sensory adjustment. The sensor can be tilted up to 27◦ either up or down.

## 4.1.2 GPU

|  |  |
| --- | --- |
|  | **GeForce GTX 1050 (4 GB)** |
| **GPU Architecture** | Pascal |
| **NVIDIA CUDA® Cores** | 640 |
| **Frame Buffer** | 4 GB GDDR5 |
| **Memory Speed** | 7 Gbps |
| **Boost Clock** | 1455 MHz |

Figure 24: GPU specifications

## 4.2 Software:

The Kinect software is OpenNI and Microsoft SDK for Kinect. The table showing a comparison between OpenNI and Microsoft SDK:

|  |  |  |
| --- | --- | --- |
|  | **OpenNI** | **Microsoft SDK** |
| **Camera calibration** | ✔ | ✔ |
| **Automatic body calibration** | ❌ | ✔ |
| **Standing skeleton** | ✔  (15 joints) | ✔  (20 joints) |
| **Seated skeleton** | ❌ | ✔ |
| **Body gesture recognition** | ✔ | ✔ |
| **Hand gesture analysis** | ✔ | ✔ |
| **Facial tracking** | ✔ | ✔ |
| **Scene analyzer** | ✔ | ✔ |
| **3-D scanning** | ✔ | ✔ |
| **Motor control** | ✔ | ✔ |

Figure 25: Comparison of openNI and Microsoft SDK

These Kinect software development kits enable developers to develop applications that supports various sensors from the Kinect module and use the output from sensors in their applications.

## 4.3 Future Discussion

In the future research more high performing Neural Network architectures can be developed. The Neural Network can consist of attention mechanism because attention is all we need (Vaswani, 2017). Multiple feature engineering techniques to be developed for higher accuracy. The neural network activations at each layer and Saliency map could be used in order to exploit the better features. Saliency map can give information regarding the importance of features in the input. Hence more advanced feature engineering techniques can be applied and compared.

# Bibliography

# References

References to any book, journal paper, or website should properly be acknowledged, For example.

Ge L, L. H. (2019). Real-Time 3D Hand Pose Estimation with 3D Convolutional Neural Networks. *IEEE Trans Pattern Anal Mach Intell*, 956-970.

Guo, H. (2017). Region Ensemble Network: Improving Convolutional Network for Hand Pose Estimation. IEEE International Conference on Image Processing (ICIP).

Liuhao Ge1, H. L. (2017). 3D Convolutional Neural Networks for Efficient and Robust. (p. 9). 2017 IEEE Conference on Computer Vision and Pattern Recognition.

Markus Oberweger, P. W. (2020). Generalized Feedback Loop for Joint Hand-Object Pose Estimation. *IEEE Trans. Pattern Anal. Mach. Intell. 42, 8 (Aug. 2020).*

Rehg, J. M. (1994). Visual tracking of high DOF articulated structures: An application to human hand tracking. *Eklundh JO. (eds) Computer Vision.*

Tompson, J. (2014). Real-Time Continuous Pose Recovery of Human Hands Using Convolutional Networks. *ACM Trans. Graph. 33, 5, Article 169 (August 2014),*, (p. 10).

Tran, D. (2018). A Closer Look at Spatiotemporal Convolutions for Action Recognition. *CVPR.*

Vaswani, A. (2017). Attention Is All You Need. *31st Conference on Neural Information Processing Systems (NIPS 2017).* Long Beach, CA, USA.