An OpenCL Based Heterogeneous Server Simulation for Edge-AI Applications

Dheemanth Joshi

Centre for Innovation and Entrepreneurship
PES University Bengaluru Karnataka- 560085

Abstract: With increased employment of data and compute intensive Artificial Intelligence and Machine Learning (AI/ML) algorithms, The demand of High-Performance Computing (HPC) has significantly increased over the last decade. AI/ML based tasks are process intensive, which is power consuming and suffer high latency with respect to a general-purpose processor. To overcome this issue, Application Specific Integrated Circuits (ASICs) such as Graphic Processing Units (GPUs), Vision Processing Units (VPUs), Field Programmable Gate Arrays (FPGAs) are introduced in modern servers. Modern day edge servers are equipped with state-of-the-art ASICs and hold the capability to process a lot of requests parallelly. However, resources installed on the server are underutilized in many scenarios due to inefficient scheduling. In this report we explore various types of scheduling schemes used in latest research and industry projects. A detailed study on Intel's Developer's Cloud for Edge AI applications is provided. Finally, we use OpenCL framework to implement an Edge Server Simulation integrated with Weighted Round Robin (WRR) based task scheduler to control the CPU+GPU Heterogenous Platform.

Methodology: We follow a structured methodology for our project which begins with a thorough understanding of the concepts involved in our problem statement. First, we study Intel's Edge AI eco system. Next, we present a detailed review summarizing the techniques which exist in the literature, we also present our understanding on OpenCL

framework. Finally, we implement a simple server platform simulation on Dell Precision 5520 workstation.

Learning Outcomes of the Internship:

1) **Working of the Intel DE-10 SoC FPGA**

2) **Brief Understanding on Intel Distribution of OpenVINO framework. (Source: Intel's Edge AI certification).**

3) **Improvised C/C++/Python coding skills for Embedded Development.**

4) **Strong Foundations in OpenCL Framework (C99 and Host API).**

5) **Foundations in Machine Learning Based Scheduling schemes.**

Table of Contents

Introduction

High Performance Computing (HPC) has evolved over the last decade possessing the capability to perform almost quadrillion instructions per second which outperforms a conventional 3 GHz Processor which can process 3 billion calculations per second. HPC framework comes with thousands of compute units which parallelly execute multiple tasks at the same time. With the demands of cloud edge computing ever increasing, HPC framework plays an important role to satisfy millions of users who use the benefits of the same.

With recent developments in ASIC and Hardware Accelerators, cloud and edge servers are integrated with Heterogenous Computing cores such as variable frequency CPUs, GPUs, FPGAs, DSPs etc.... which process application specific tasks such as Internet of Things (IoT) and (AI/ML) applications with accelerated speed. This provides a great platform for latency sensitive applications to run at the edge. Autonomous Driving is one of the examples for latency sensitive applications where the tasks provided by the customer should be performed and delivered in a highly dynamic and stringent environment. This makes integration of HPC and Edge crucial.

With this regard, we explored Intel's Developer Cloud and OpenVINO Framework to understand the real time implementation of heterogenous platforms at the edge. We realized the key concepts to use the framework provided by the vendor and use it for our research. Detailed working of OpenVINO is explained in later sections.

We also had the opportunity to explore Intel DE-10 standard board and realize its applications at the edge. DE-10 board is integrated with dual core CPU and a FPGA which makes it a perfect device for heterogenous edge computing. More details on the DE-10 Board are provided in later sections.

Although edge compute devices integrated with heterogenous cores are making significant improvements in performance, the hardware resources installed on the server go underutilized due to inefficient scheduling schemes. In this project, we explore the role of Machine Learning (ML) based task schedulers to extract the features of the task and assign it to the compute such that it faces minimum delay at a given instance. We also implement an OpenCL based heterogenous platform on which the schedulers can be tested for future work.

OpenCL provides ideal environments to conduct heterogenous computing experiments. OpenCL introduces OpenCL “Kernels” which can execute on any vendor supported device which provides a common platform to run the tasks. Kernels can be scheduled using Host API which acts like the controller for all the connected devices. More on OpenCL is discussed in later sections.

A Brief Description and Working of DE-10 Board

Intel DE-10 standard is a robust platform which consists of dual core ARM Hard Processing System (HPS) and a cyclone FPGA. This is a perfect setup for high performance and low power computing. Visual representation of the board is provided below.

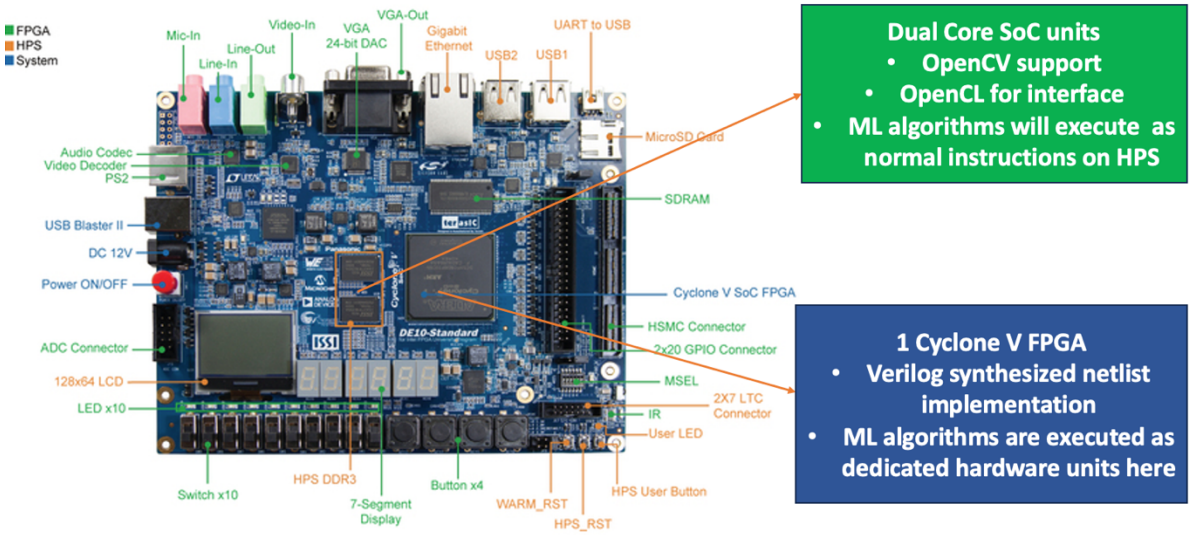

Fig-1: DE-10 Board peripherals

With built in OpenCL Board Support Package (BSP), we can enable high performance heterogenous computing on the Edge as this board is light weight and is ideal for this application.

Low power applications which can run on this SoC FPGA were also explored. These applications included Edge computing, Image processing using OpenCV and Digital Design using Verilog.

Features and Tools of the Board:

Configuration of the board is set using MSEL pins. It is crucial to set up the pins as directed by the user manual of the board failing which may

cause catastrophic affects to the board. The configuration is shown in Figure 2.

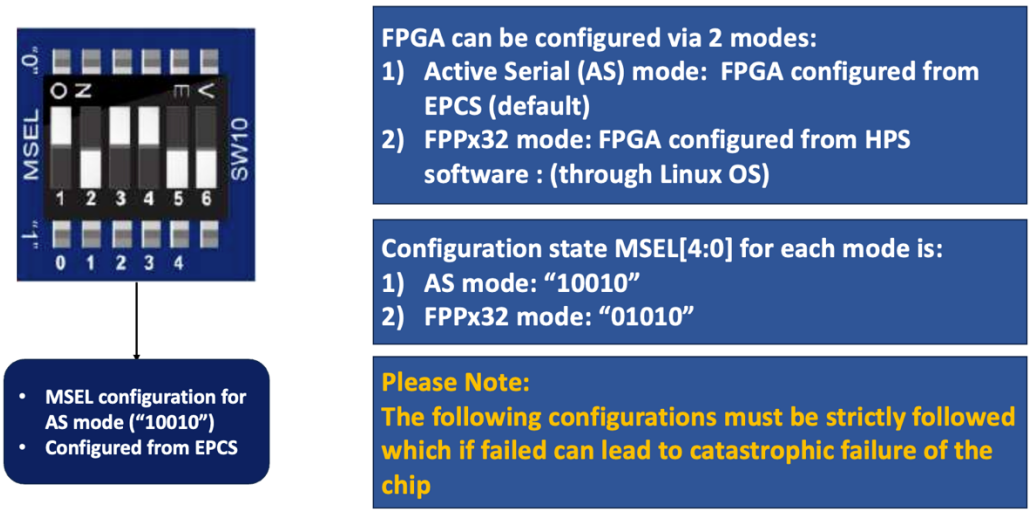

Fig-2: MSEL Configuration Information

1) **Operating System**: which runs the board is a Linux based LXDE environment which comes with GUI and control panel which can be used to control various sensors on the board. A screen shot of the GUI is shown below.

Fig-3: Screen Shot of the LXDE Environment

It can be seen in Figure-3 that the board comes with some pre-built executable files which control the embedded sensors on the board.

However, appropriate software can be configured to control the sensors for application specific processes.

2) **Ports and Interfaces**: The board is equipped with USB ports, Ethernet port and VGA ports for communication. Most of these ports are directed to the HPS for routine protocols. However, video and audio ports such as TV Decoder and Audio CODEC are by default connected to the FPGA for direct accelerated computing. These ports can also be directed towards the HPS through the I2C multiplexer as shown in Figure-4.

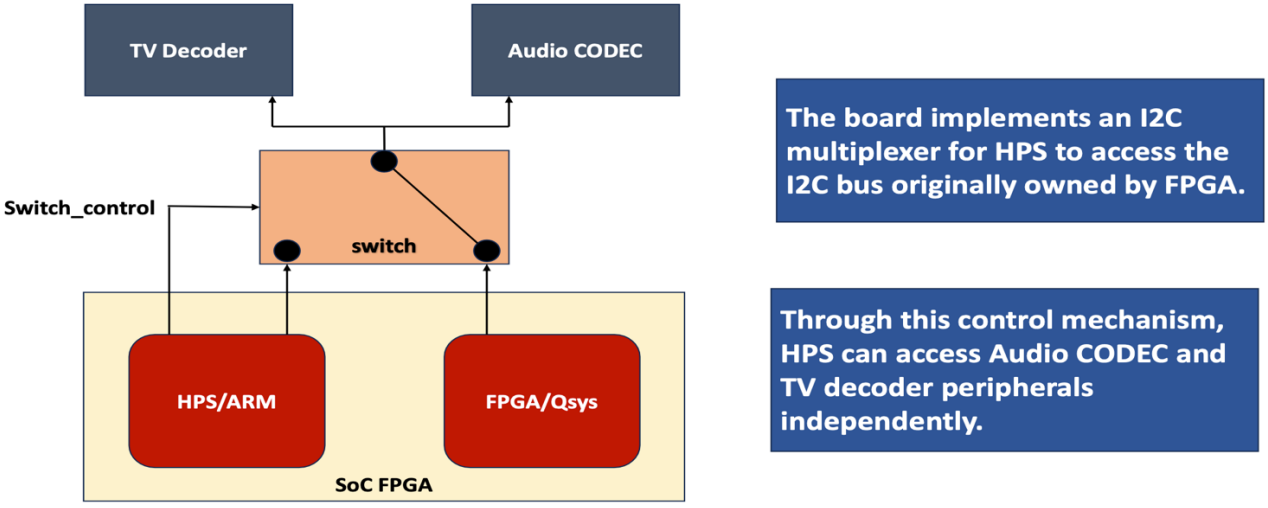

Fig-4: Working of the I2C Multiplexer

3) The SoC can be controlled from any PC. The workflow is provided by intel's SoC EDS (Embedded Development Suite) which builds C files for the appropriate ARM architecture installed on the board. The Software development flow is demonstrated in Fig-5.

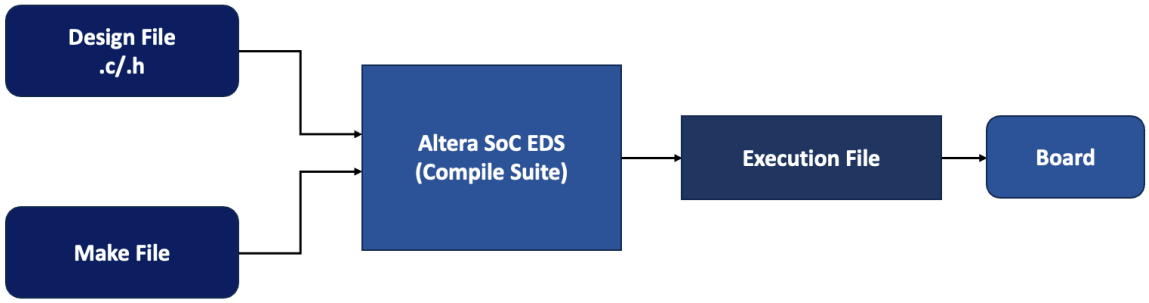

Fig-5: Software Development Flow for HPS Execution.

Once the program is built, it is transferred to the board via RJ-45 Ethernet cable. PuTTY API is used to call SSH protocol for file transfer. The framework is demonstrated in Figure-6.

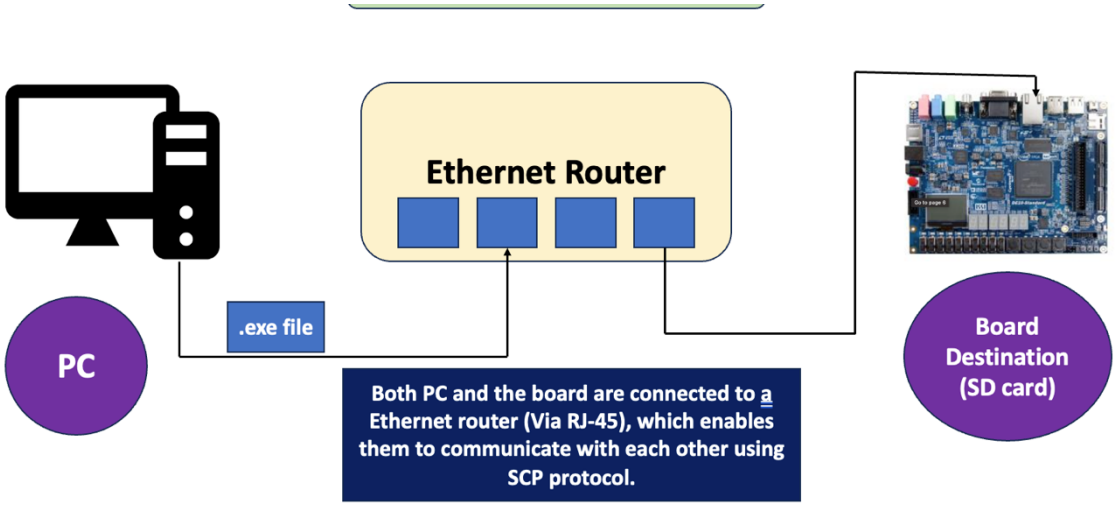

Fig-6: Setup for File Transfer

4) HPS can directly be accessed through the LXDE GUI interface and appropriate routines can be executed on the board if required resources are available.

5) The **Cyclone FPGA** can be configured in 2 ways:

   a) Quartus Prime Design Suite for custom Digital Design

   b) OpenCL/ oneAPI framework for heterogenous environments with HPS as the host

   Developers can choose the design flow based on the projects they are working on.

Summary: DE-10 is a low-power SoC FPGA with broad array of communication ports and sensors making it perfect for EDGE-AI applications.

OpenVINO for Edge AI Applications

Intel Distribution of OpenVINO provides a systematic approach to run Deep Neural Networks (DNNs) on any supporting Intel Hardware. This provides an efficient and accelerated platform to run AI applications on heterogenous platforms at the Edge.

Figure-7 provides the architecture of the OpenVINO framework.

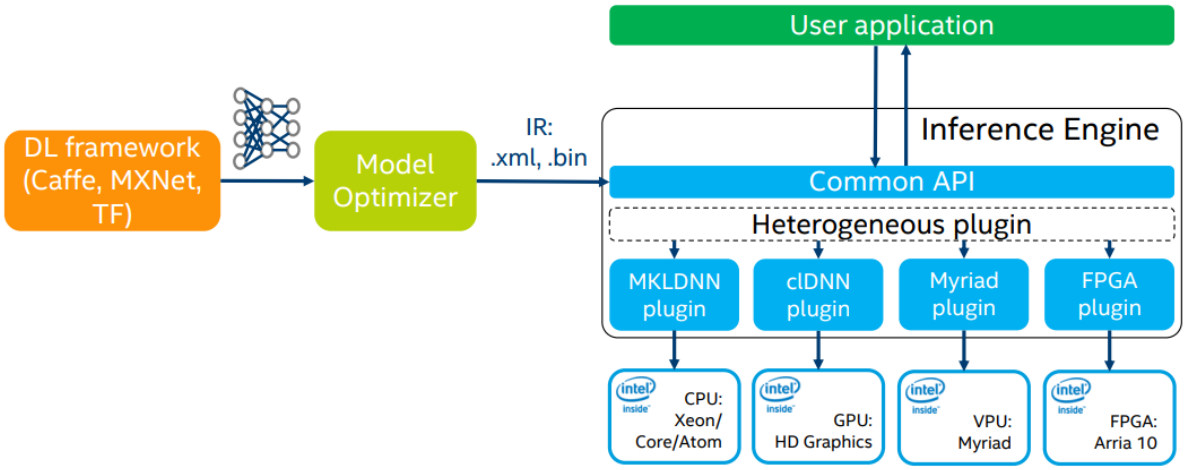

Fig-7: OpenVINO Framework

DL Framework: is the application specific DNN which is trained using Tensorflow, caffe, Pytorch etc… The trained parameters are then fed to the model optimizer to aid further processing.

Model Optimizer: The model optimizer takes in the pre trained network and converts it to a Immediate Representation (IR) file. This is a common file which is sent to the assigned device plugin through the Common API. Before this process, Model optimiser optimises the neural network model by applying common optimization techniques which results in more efficient computing in the target device.
The pre trained model can also be quantized with NNCF tool flow which reduces the computations conducted on the device.

Inference Engine: The Inference engine is a software system consisting of Common API and various heterogenous plugins connecting to the underlying hardware platform. The IR files are fed to the common API which runs the user specific applications on the host. With target device known, the IR files are immediately scheduled on to the corresponding plugins of the target device. The plugin performs the necessary transforms to the IR files and executes the network on the target device. Inference engine is constantly connected to the application which executes the DNNs on these devices.

Intel's DevCloud is an ideal resource to execute OpenVINO applications. The cloud comes with latest intel hardware which can be tested by the developers. Jupyter notebooks can be run on this cloud making it more user interactive.

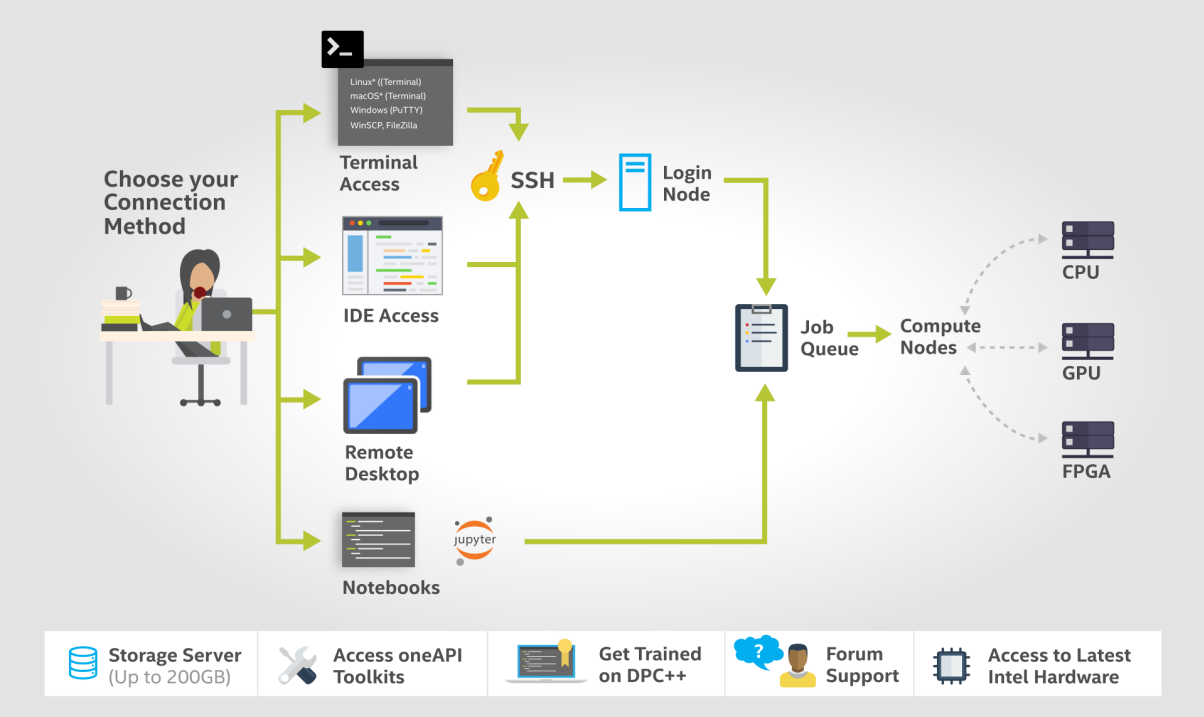

Fig-8: Overview of Intel's Developer Cloud server

OpenCL- A Brief Introduction to Heterogenous Computing

OpenCL stands for Open Computing Language and is used to interact with heterogenous systems through a common language. Heterogenous computing has become a key feature in high performance computing. With the evolution of Graphics Processing Units (GPUs), Vision Processing Units (VPUs) and Field Programmable Gate Arrays (FPGAs), these heterogenous compute cores provide application specific computing capabilities which accelerates the process. With latency sensitive applications such as Autonomous Driving Assistance Systems (ADAS) becoming mainstream in the workload of Edge servers, acceleration of routines becomes crucial.

As described earlier, OpenCL provides a common platform to interact with any class of hardware device with same language. This improvises code efficiency, provides portability and focus is directed towards innovation. In this project, we will be using OpenCL for interacting with the hardware.

OpenCL Environment and Structure:

Some structures of the OpenCL environment are listed below:

Host: Is the controller and in-charge for resource allocation and memory management. The task is offloaded to the target device through the host. The host is the lone manager of memory, task offloading and scheduling. This structure is followed to give the developer full access to the resources and memory, and he/she can optimize it according to his/her application.

Device: Is the heterogenous compute node connected to the host. Each device is structured to have compute units each having multiple processing elements which execute the tasks offloaded to the device.

Host API: The programming language which performs the functionality of the host (offloading tasks to the device and scheduling). Host API can run only on a CPU. Host API used currently are C, C++ and Python.

Kernel: Is a C99 based device function which is the task to be executed by the device. Kernel compilers are specifically written by the vendors for personal platforms. For example, a OpenCL kernel will be compiled to a CUDA code by Nvidia compiler.

Work Item: Is a specific task executed by a processing element in the device. Work item generally corresponds to single kernel code written for the device.

Work Group: Collection of work items is called a work group. This group is executed generally on 1 compute unit. Synchronization is only possible within work groups. All the work groups are executed parallelly on the compute units of the device.

Platform: Platform corresponds to a list of devices which belong to a specific vendor. For example, Intel's platform on Dell Precision 5520 workstation contains an octo-core processor and an integrated graphics card.

Command Queue: Command queue maps the tasks to the target device. One command queue maps to one device. But multiple command queues can port to single device. Kernel calls, Host - Device memory copy calls etc… are the routines enqueued to the command queue.

Buffers: Buffer object corresponds to device memory in the host. Buffers hold the necessary data which is used by the kernel during the compute phase. There are memory operations provided by the OpenCL framework which allows us to read, write and copy data between buffers and host memory.

Context: Context binds command queues, devices, buffers and tasks to create the OpenCL system. Multiple devices from various platforms can be bound to same context and parallel computing is enabled by the OpenCL compiler. This defines the heterogenous parallel computing environment which can be deployed at the edge.

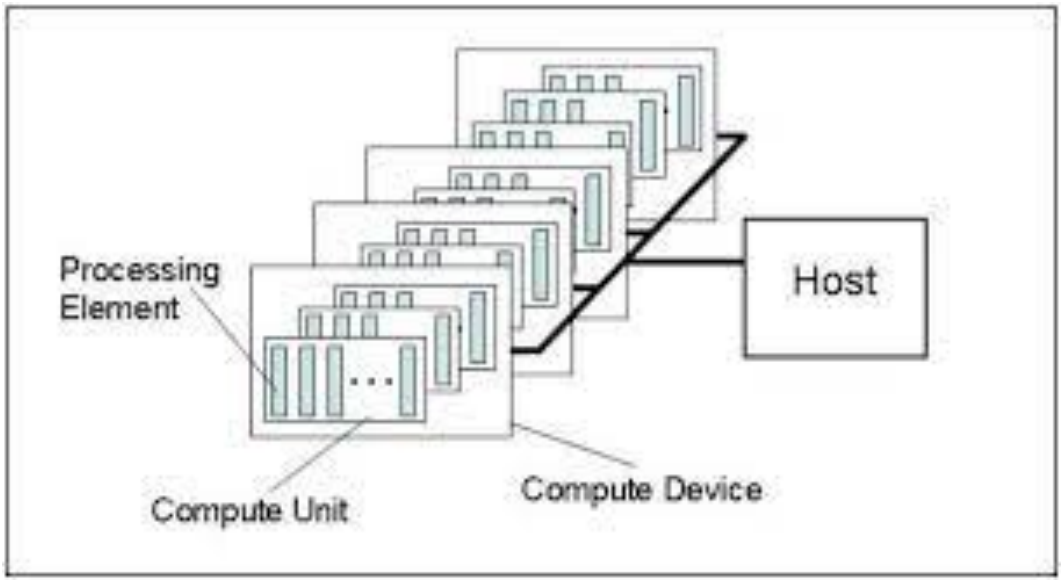

Fig-9: Host-Device Architecture for OpenCL

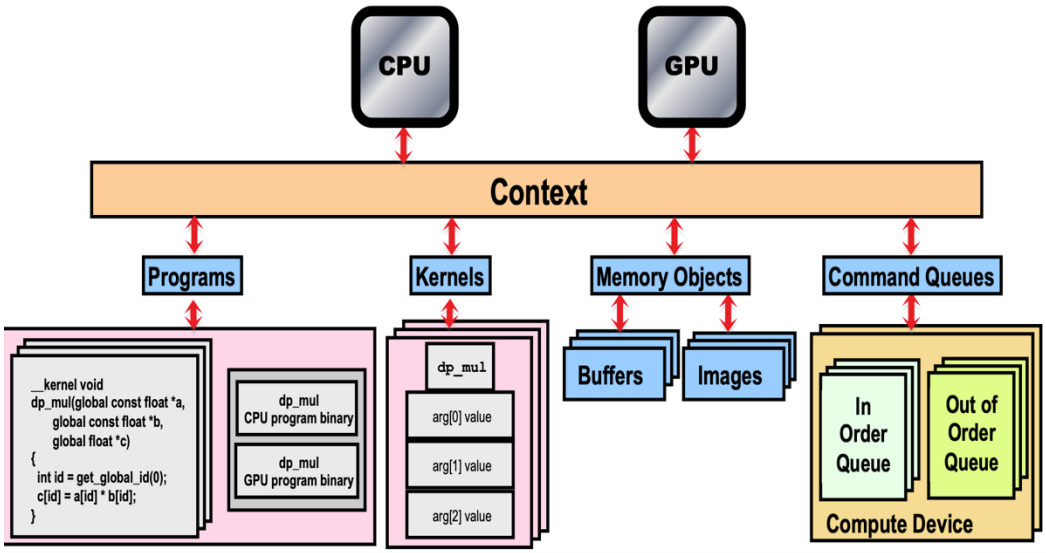

Fig-10: Overview of the OpenCL Framework

OpenCL In Action: A simple simulation.

In this sub-section, we will be simulating a simple OpenCL code which performs vector addition on Nvidia GPU.

System Requirements:

1) Python 3 with pyopencl library and other common imported libraries
2) Dell Precision 5520 workstation with Nvidia Quadro 1200 GPU.

Methodology:

1) Import numpy and pyopencl libraries

```
In [1]: import pyopencl as cl
        import numpy as np
```

2) Set up the platform and devices.

```
In [3]: """Set up platforms and devices"""

        cl.get_platforms()

        platform_nvidia=[platform for platform in cl.get_platforms() if platform.name=="NVIDIA CUDA"][0]

        platform_intel=[platform for platform in cl.get_platforms() if platform.name=="Intel(R) OpenCL HD G

        devices_nvidia= platform_nvidia.get_devices()

        devices_intel= platform_intel.get_devices()

        print(devices_nvidia, devices_intel)

        [<pyopencl.Device 'Quadro M1200' on 'NVIDIA CUDA' at 0x28ded1b5660>] [<pyopencl.Device 'Intel(R) HD Graphics 630' on 'Intel(R)
        OpenCL HD Graphics' at 0x28dee88a150>]
```

3) Create context and build the programs for specific devices.

```
In [4]: """ Create Context and build the program"""

        programs= []

        ctx_nvidia= cl.Context(devices= devices_nvidia)

        ctx_intel= cl.Context(devices= devices_intel)

        prg_n= cl.Program(ctx_nvidia, kernel).build()

        prg_i=cl.Program(ctx_intel, kernel).build()

        programs.append(prg_n)

        programs.append(prg_i)
```

4) Create host memory using numpy arrays.

```
"""Host Memory"""
a_h= np.array([i for i in range(1000)], dtype=np.float32)

b_h= np.array([i for i in range(1000)], dtype=np.float32)

c_h_n=np.empty_like(a_h)

c_h_i=np.empty_like(a_h)
```

5) Create Buffers for Nvidia GPU

```
"""Device Memory- nvidia GPU"""

a_d_n= cl.Buffer(ctx_nvidia, flags= cl.mem_flags.READ_ONLY, size= a_h.nbytes)

b_d_n= cl.Buffer(ctx_nvidia, flags= cl.mem_flags.READ_ONLY, size= b_h.nbytes)

c_d_n= cl.Buffer(ctx_nvidia, flags= cl.mem_flags.WRITE_ONLY, size= c_h_n.nbytes)
```

6) Create command queue and enqueue Host-device memory copy operation to the nvidia command queue.

```
""" Command Queue definition """

queue_nvidia= cl.CommandQueue(ctx_nvidia)

queue_intel= cl.CommandQueue(ctx_intel)

""" Copying Host memory inputs to device memories  """

cl.enqueue_copy(queue_nvidia, src= a_h, dest= a_d_n)

cl.enqueue_copy(queue_nvidia, src= b_h, dest= b_d_n)

cl.enqueue_copy(queue_intel, src=a_h, dest=a_d_i)

cl.enqueue_copy(queue_intel, src=b_h, dest=b_d_i)
```

7) Run the Job on device

```
In [6]: """ Run the jobs on devices """

        programs[0].vadd(queue_nvidia, (1000,), (10,), c_d_n, a_d_n, b_d_n)

        programs[1].vadd(queue_intel, (1000,), (10,), c_d_i, a_d_i, b_d_i)

        cl.enqueue_copy(queue_nvidia, src=c_d_n, dest= c_h_n)

        cl.enqueue_copy(queue_intel, src= c_d_i, dest= c_h_i)

        print(c_h_n[:10])

         [ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18.]
```

The result of adding 2 whole number vectors is printed in this segment.

Kernel used for vector addition is shown below.

```
kernel="""

__kernel void vadd(__global float *c , __global float *a, __global float *b)
{

int gid= get_global_id(0);

c[gid]=a[gid]+b[gid];

}

"""
```

Resource utilization and memory management is left to the developer in OpenCL framework as demonstrated in the above simulation. However it is challenging to for any system developer to efficiently utilise the enormous computing resources available on the platforms. Hence Task Scheduling in OpenCL based systems is proves to be a significant problem in the domain.

Literature Survey: Task Scheduling on OpenCL based Environments.

Detailed literature review was conducted to get familiar with the latest scheduling techniques which are used in industries and research. Summaries of the same are provided below.

1) **Paper 1**: Predictive Dynamic Scheduling Approach for Heterogenous Resource Pool

   Description: Authors attempt a dynamic scheduling approach for job scheduling in GPU cluster-based environments. The core of this problem lies in predicting the compute usage characteristics of each job for a given period then schedule it on the appropriate device.

   Methodology: Traditional scheduling algorithms such as FCFS (First come first serve), BestFit, shortest job first etc… are described in the paper. Role of AI has been explained in the domain of task scheduling algorithms. Authors tend to compare their approach with the existing algorithms.

   System Description:

   a) Monitor the runtime resource usage of the application (Variational autoencoder, DCGAN etc…) on the GPU clusters.

   b) Create a dataset with features and targets mentioned above.

   c) Train a non-linear high dimensional model for scheduling.

   d) When a new job enters the queue, it can predict its resource utilization from the trained neural network.

2) **Paper 2**: Feature Aware Task Scheduling on CPU-FPGA Heterogenous Platforms.

Description: Authors present a novel feature aware task scheduling approach for CPU-FPGA heterogenous platforms. This scheme demands in depth feature study of OpenCL kernels which is used to predict speedup using ML techniques. Finally, a scheduler is designed which takes task size and classifier's opinion to schedule the task on any one device.

Methodology: a brief feature analysis of the incoming tasks is conducted. Features such as static features of the algorithm, algorithm complexity, bandwidth of the CPU-FPGA bus etc… are extracted. A Support Vector Machine, (SVM) classifier is trained to learn these features and predict the speedup threshold of each OpenCL kernel.

System Description:

The system model is demonstrated below, (Figure 11)

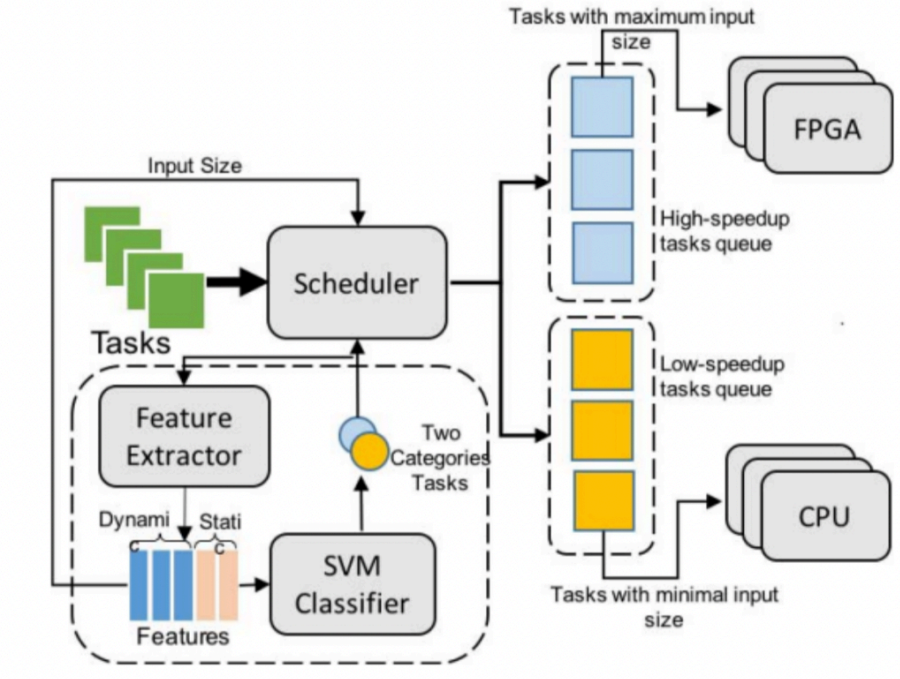

Fig-11: System model of Paper 2

3) **Paper 3**: Smart Multi-Task Scheduling for OpenCL Programs on CPU/GPU Heterogeneous Platforms

Description: A smart task scheduling scheme is designed to map multiple OpenCL kernels coming from multiple applications. This is achieved by determining the best device for a particular OpenCL kernel keeping resource utilization as the key distinction feature.
Methodology: Build an offline predictor using training programs. The built model can then be used within OpenCL task scheduler. Program features and respective speedups are used to train the Machine learning model. Kernels used for task scheduling include bfs, Dotproduct, QuasirandomG etc…

System Description:

1) Multiple kernels arrive from multiple applications to the host.
2) Pre trained offline model predicts the approximate speedup achievable for a given kernel task.
3) If the speedup is greater than a given threshold, the kernel is scheduled on a GPU else, it is scheduled on a CPU.

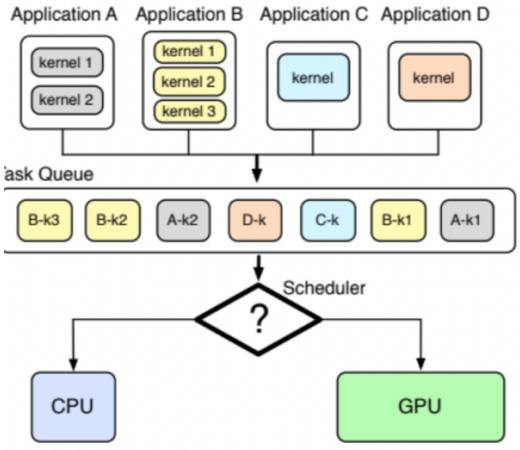

Fig 12: System Model (P3).

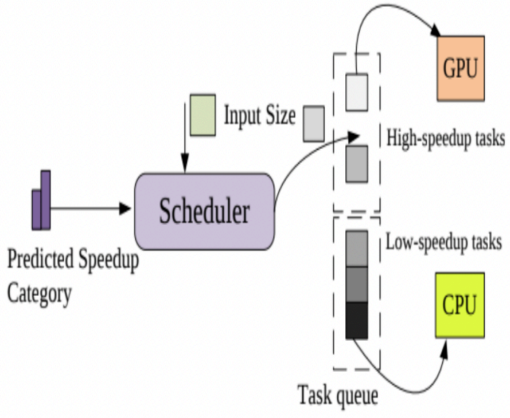

Fig-13: Scheduler design (P3)

System Model

In this section we will discuss the system model for OpenCL based heterogenous environment which can be deployed on the edge. The purpose of this model is to demonstrate the OpenCL tool flow and integration between scheduler and hardware platform. Mainly, our system is divided into 2 parts:

1) **Hardware Platform setup**

2) **Task Scheduler Design**

1. **Hardware platform setup**: To simulate the server environment, we have to setup the heterogenous hardware platform. To achieve this, we use OpenCL framework. We create a heterogenous environment consisting of intel i7 7th gen octo core processor, intel integrated graphics card and Nvidia Quadro M1200 GPU.
2. **Task Scheduler Design**: We implement a simple Finite State Machine (FSM) based task scheduler to schedule the incoming tasks. The FSM will perform a Weighted Round Robin (WRR) based approach to allocate tasks to multi device platform. The scheduler will be implemented on host software. i.e., the CPU will run the scheduler.

Overview of the system model is displayed in figure 14:

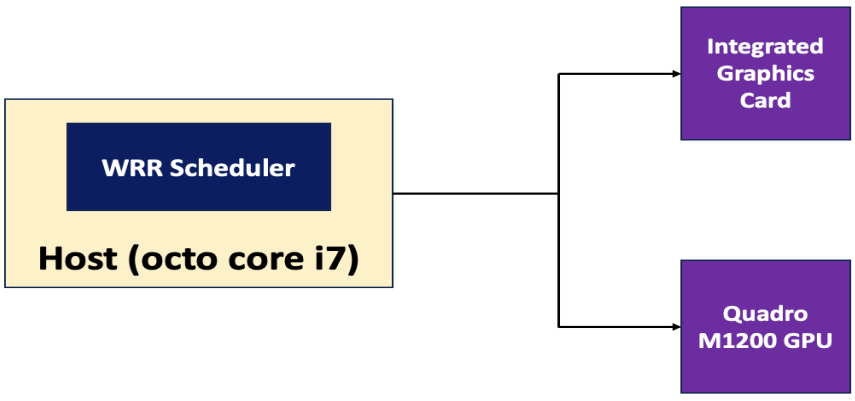

Fig 14: System Model

Implementation: Setting up the Hardware Platform

We begin our implementation by setting up our hardware platform and get control of it through software. We achieve this by using some of the built routines provided by OpenCL. Simulations was run on Dell Precision 5520 Workstation. More details about the platform are given in figure 15.

```
Server Description:

0) Server System: Dell Precision 5520 series with 32 GB RAM and 2Tb SSD

1) Devices available:

    Nvidia Quadro GPU- 1

    Intel Graphics Card- 1

    Intel octacore i7 7th gen processor- 1

    (1 processor to be used as host)

2) Request format:

    Requests come as tasks with a specific architecture.

3) Tasks:

     (Image processing and ML oriented kernels with accurate input and output dimension
description)

     Input data and output data shape

     task dependent data description
```

Fig-15: Server Simulation Setup

To set up the platform, following steps were taken:

1) Import the following libraries shown in Figure 16.

```
In [2]: import numpy as np

        import pyopencl as cl

        import matplotlib.pyplot as plt

        import random
```

Fig-16: Libraries to be imported for simulation.

2) Call “cl.get_platforms” to obtain all platforms available in the workstation.
3) Obtain all the devices available in Intel and Nvidia platforms by using “platform.get_devices”. In our case, we get 1 integrated graphics card and 1 Quadro M1200 GPU.

4) We also yield the number of compute units present in each device. We achieve this by calling “device.get_info” method.

The above steps were executed to set up the platform. The same is illustrated in Figure-17

```
In [3]: """ Set up the server's platform and devices """

        platforms= cl.get_platforms()

        platform_GPU0= [platform for platform in platforms if platform.name== "Intel(R) OpenCL HD Graphics"

        platform_GPU1= [platform for platform in platforms if platform.name== "NVIDIA CUDA"][0]

        platform_CPU= [platform for platform in platforms if platform.name== "Intel(R) OpenCL"][0]

        devices_GPU0= platform_GPU0.get_devices()

        devices_CPU= platform_CPU.get_devices()

        devices_GPU1= platform_GPU1.get_devices()

        devices_GPU1[0].get_info(cl.device_info.MAX_COMPUTE_UNITS)
```

Fig-17: platform setup using pyopencl

To interact with the device, we follow the OpenCL framework which is listed below:

1) **Create Context**:
   We create two separate contexts for the two graphics cards present in the platforms. The structure is demonstrated in Figure-18.

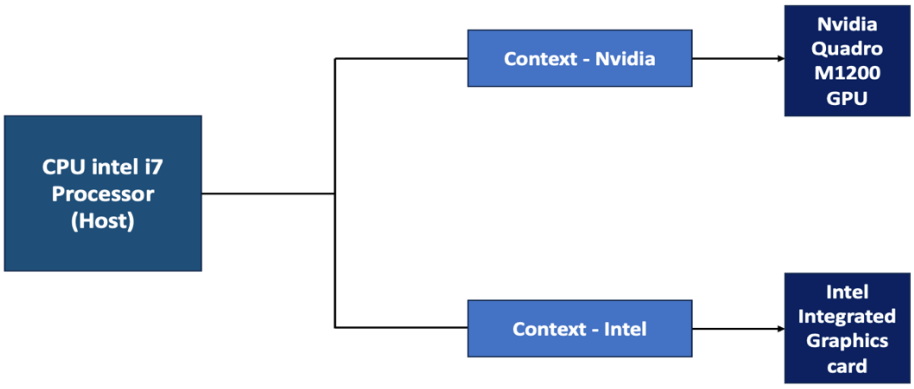

Figure-18: Host-device bound with the context.

2) **Create Command Queue**: we create 3 parallel command queues for each device. This enables OpenCL compiler to employ parallel scheduling and tasks get executed on multiple compute units on the device parallelly

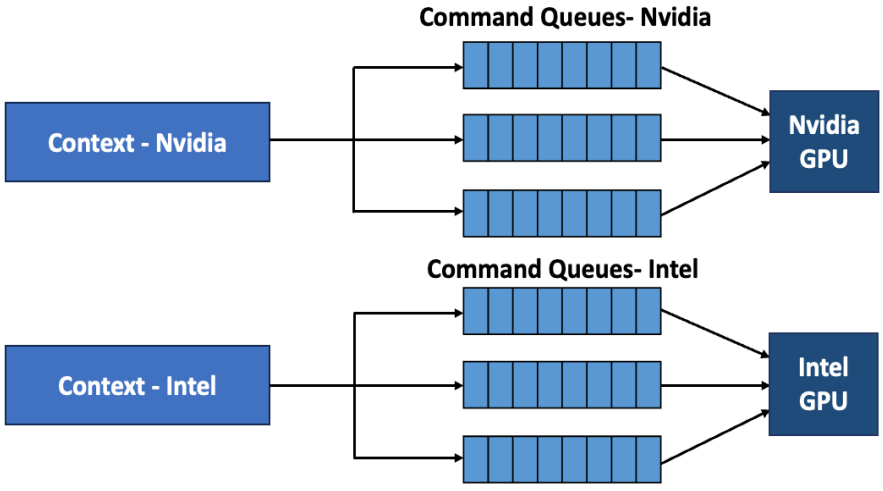

Figure-19: Command Queue Architecture

3) **Buffers to store task parameters**: we create buffers at runtime when the tasks arrive. Buffers map to hardware memory which is inferred on the host. The incoming tasks are copied to the buffers through the command queue and are ready to be used for kernel routines. Buffers are also bound by context.

4) **Build and enqueue the kernel on the command queue**: To run the kernel on hardware (assuming the required data is present in the buffers) we use “program.build” function. Once the kernel is built for a specific context, it can be enqueued in the command queue to be executed.

5) **Free the buffers**: Once the task is completed, the allocated buffers are freed so that the garbage collector can free the used memory. This is a critical process, failing which can lead to memory leakage.

Finally, all the models are integrated which completes our hardware platform setup.

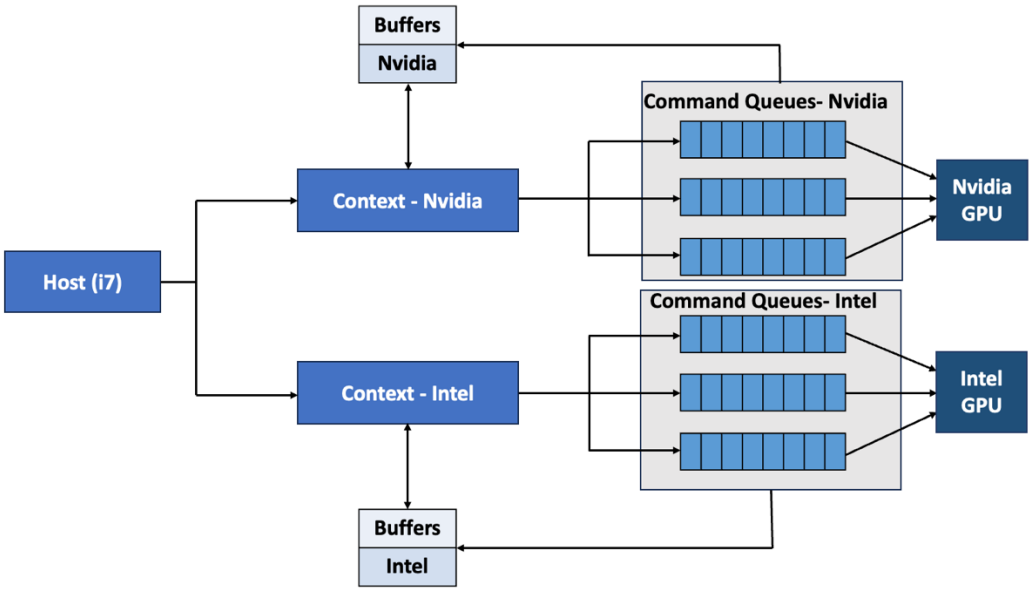

Fig-20: Overview of the OpenCL based Hardware Platform Setup

Implementation: Design of the Task Scheduler

We implement a simple “Weighted Round Robin (WRR)” method for task scheduling. By using this method, we exploit the compute capacity feature to balance the load between devices equally i.e. the device having higher compute capacity will be assigned with more tasks. This improvises resource utilization and optimizes the latency metrics.

We implement a finite state machine to implement WRR method. The implementation is demonstrated in Figure-21.

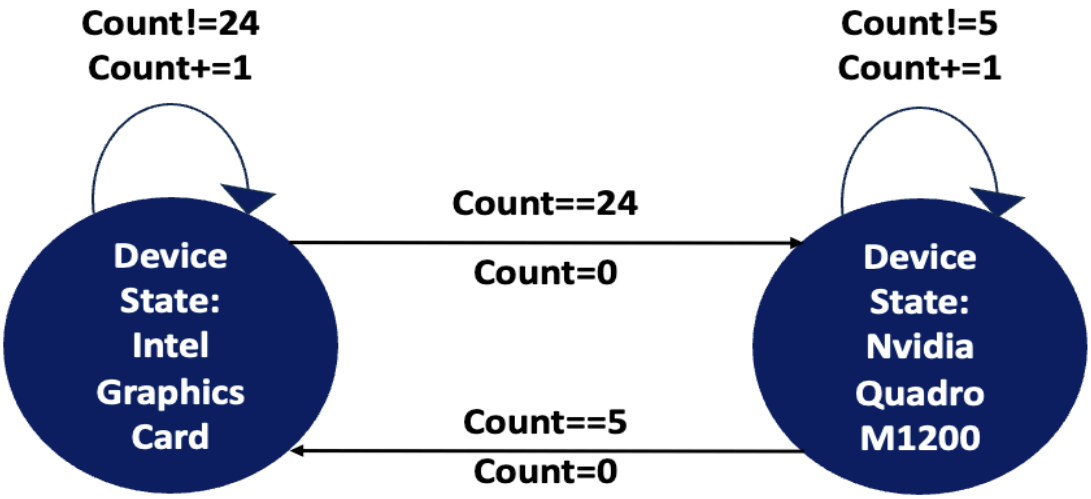

Fig-21: Overview of the FSM implementing WRR.

since the intel graphics card contains 24 compute units and Nvidia graphics card contains 5 compute units, we weigh more tasks on the intel graphics card assuming computing units from both the vendors have identical processing elements. The non-blocking nature of OpenCL framework enables us to query multiple tasks to multiple devices at once which enables parallel computing.
Finally, the task scheduler is integrated with the hardware platform. The scheduler acts like the controller to distribute kernels between the two GPUs. This is demonstrated in Figure-14.

Results

In this section, we will be demonstrating the results obtained from the server simulation.
The server was simulated for a period of 100 time slots. We present the latency and load parameters which are critical features in our problem domain.

Vector addition kernels were run on the platform. To study the load balancing features, the input task size is assumed to be equal.

1) **Latency**: For each time slot, random number of tasks were generated and were assigned to the server platform. Execution time for each task was calculated. The execution times are demonstrated in Figure-22.

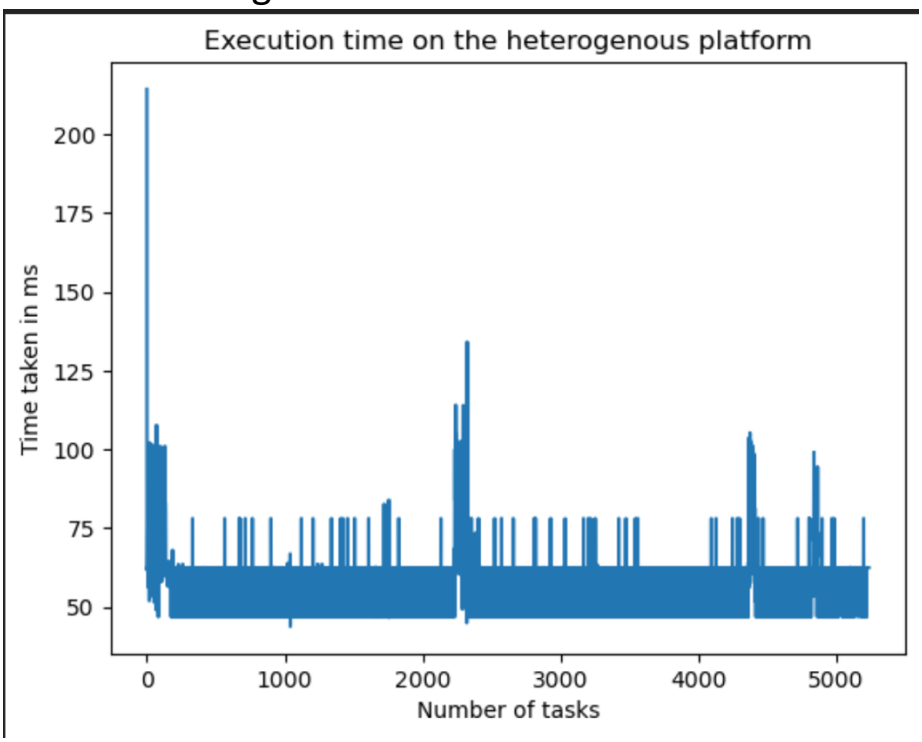

Fig-22: Execution times on the platform.

2) **Load**: Load on each device at each time slot was monitored to estimate the trend of the scheduling algorithm. A priority-based load distribution was observed through weighted round robin approach. Roughly, for 1 task allocated to Nvidia GPU, 6 tasks were allocated to the intel graphics card. Load distribution for

each device is demonstrated with following graphs in figures 23,24 respectively.

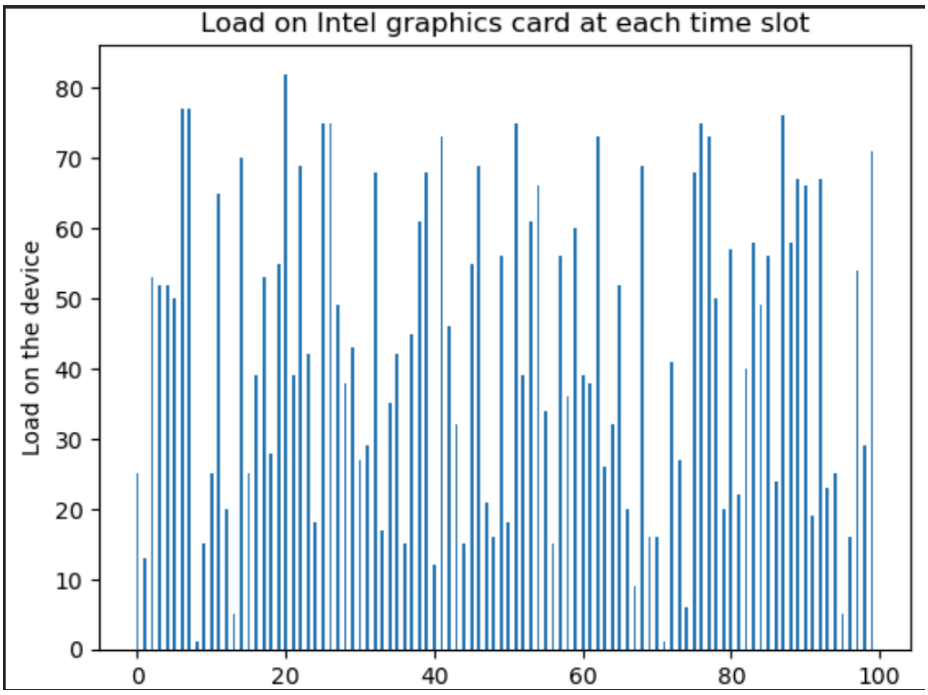

Fig-23: Load v/s time slot for Intel graphics card.

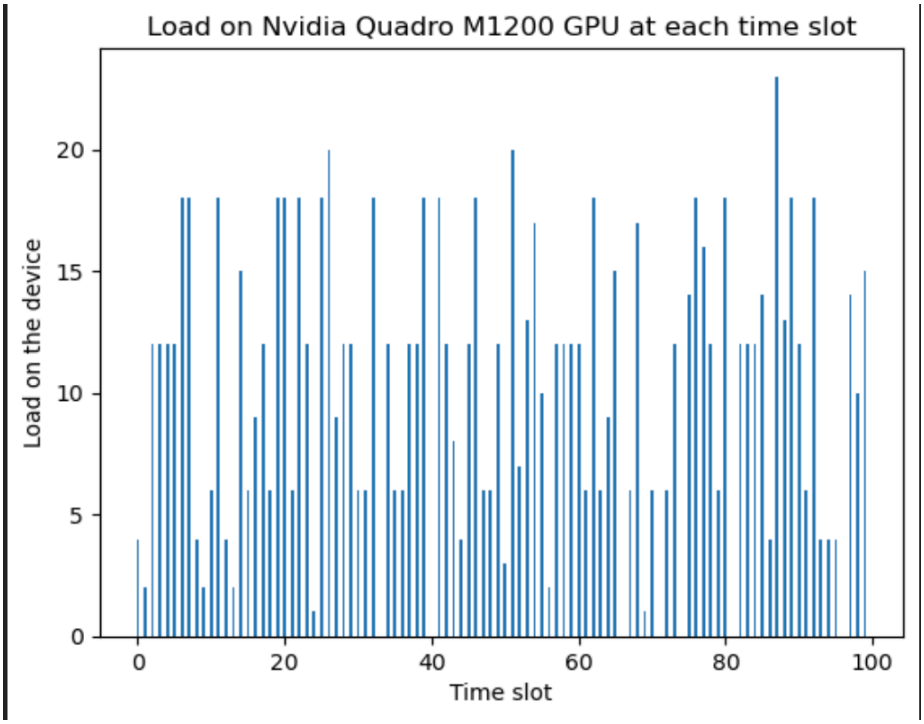

Fig-24: Load v/s time slot for Nvidia GPU

The following table (Table 1) demonstrates average latency and load distribution within the platform.

Average Latency	60.37 ms
Average Load on Intel Graphics Card	42.25
Average Load on Nvidia GPU	10.11

Conclusion and Future Work

In this project, we explored various applications and problems involved in the domain of High-Performance Computing for Edge AI applications. We started off by studying the Intel DE-10 board, which can be used as an edge device for various low power computing applications. We went through Intel's distribution of OpenVINO toolkit which enables us to execute various Deep Learning models on heterogenous intel devices. We provided a detailed survey on the OpenCL framework for heterogenous environments by demonstrating the software's framework. We conducted a detailed literature survey on task scheduling problem in heterogeneous environments for latency critical systems. Finally, we demonstrated the working of an OpenCL based heterogenous server by building a custom heterogenous hardware platform and a WRR scheduler to control the platform. Results indicated that WRR is a decent task scheduler for load balancing in multicore heterogenous system.

In the future, we aim to explore Machine Learning (ML) based task scheduling algorithms for more efficient allocation in latency sensitive systems. We also aim to accelerate the scheduling algorithm on the most suited processing hardware available to us. With the information gathered about the DE-10 board, we would like to integrate FPGAs to our heterogenous environment and test its performance.

----------XXXXXXXXXX----------