www.ijemr.net ISSN (ONLINE): 2250-0758, ISSN (PRINT): 2394-6962

Vandana Publications
IJEMR

Volume-7, Issue-3, May-June 2017
International Journal of Engineering and Management Research
Page Number: 812-817

Design of Open Computing Language Framework for Embedded Multi-Core Processors

P.Ramya[1], E.Kavitha[2]
[1]M.Tech-IInd Year [ES]- ECE, Vidya Jyothi Institute of Technology, Hyderabad, INDIA
[2]Assistant Professor, ECE, Vidya Jyothi Institute of Technology, Hyderabad, INDIA

ABSTRACT

In modern mobile embedded systems, various energy-efficient hardware acceleration units are employed in addition to a multi-core CPU. To fully utilize the computational power in such heterogeneous systems, Open Computing Language (OpenCL) has been proposed. A key benefit of OpenCL is that it works on various computing platforms. However, most vendors offer OpenCL software development kits (SDKs) that support their own computing platforms. The study of the OpenCL framework for embedded multi-core CPUs is in a rudimentary stage. In this paper, an OpenCL framework for embedded multi-core CPUs that dynamically redistributes the time-varying workload to CPU cores in real time is proposed. A compilation environment for both host programs and OpenCL kernel programs was developed and OpenCL libraries were implemented. A performance evaluation was carried out with respect to various definitions of the device architecture and the execution model. When running on embedded multi-core CPUs, applications parallelized by OpenCL C++ showed much better performance than the applications written in C++ without parallelization. Furthermore, since programmers are capable of managing hardware resources and threads using OpenCL application programming interfaces (APIs) automatically, highly efficient computing both in terms of the performance and energy consumption on a heterogeneous computing platform can be easily achieved.

Keywords-- OpenCL, energy consumption, C++, software development kits, application programming interfaces (APIs)

AIM AND OBJECTIVE OF PAPER

The main objective of the paper is to "Design of OpenCL framework for embedded multi-core processors". The aim of this paper is to get a fully functional single board computer (SBC) working with custom built monitoring software that communicates with all modern devices. It should be capable of extracting the necessary data from the control unit in order to use it in a meaningful and useful way. Communication to and from the CU will be done using the Onboard Diagnostics.

I. INTRODUCTION

Modern embedded consumer devices are often required to process multiple computationally intensive applications simultaneously. Therefore, various application -specific accelerators are commonly employed. Typically, mobile devices are equipped with a multi-core central processing unit (CPU), a multi-core graphics processing unit (GPU), digital signal processors (DSPs), image signal processors (ISPs) and video decoders. Correspondingly, a set of software frameworks is ported to utilize such devices. Since these processors have distinct hardware structures and instruction sets, compilers should generate the corresponding target binary codes to utilize them. Once a certain binary code for a particular target is generated, the binary code can be executed only by the specific target device.

Thus, it is not easy to fully utilize the various computing resources at runtime. Therefore, it is desirable to have a flexible and dynamically redistributable computing environment. By having such an environment, dynamic job assignment and load balancing become feasible, and better real- time responsiveness and reduction in the power consumption can be achieved.

Open Computing Language (OpenCL) was proposed to provide a framework that supports heterogeneous computing platforms. OpenCL includes programming languages and application programming interfaces (APIs). OpenCL C is the programming language to write a parallel function code called the OpenCL kernel. When the kernel is compiled dynamically, it can be executed on various heterogeneous devices [2].

However, unlike the original motivation of OpenCL, most vendors offer OpenCL software development kits (SDKs) that support their own computing platforms [3], [4]. Especially, for embedded mobile devices, communication interfaces and APIs may not be openly available to developers, and therefore, it is not easy for programmers to efficiently utilize heterogeneous computing platforms. Hence, it is necessary to develop an OpenCL framework that automatically generates binary codes from OpenCL codes for various computing devices. By using this

framework, heterogeneous computing platforms can be utilized by a program written in one language, which is the key advantage of OpenCL.

Various ways of executing parallelized application programs on computing devices have been actively studied [5]-[5]. Each computing resource has pros and cons in terms of the performance, applicability, and power consumption. GPUs are known to be effective in dealing with data parallelism, while CPUs are better for task parallelism. Park *et al.* exploited both data parallelism and task parallelism in designing a parallel low-density parity-check (LDPC) decoder in such a way that some parts that were more appropriate for data parallelism were assigned to the GPU, and others that were more appropriate for task parallelism were assigned to the CPU [8].

However, as the number of applications running simultaneously on an embedded device increases, the characteristics of the workload vary significantly. Therefore, it However, as the number of applications running simultaneously on an embedded device increases, the characteristics of the workload vary significantly. Therefore, it is highly desirable to have a flexible OpenCL framework that can redistribute workloads at runtime to fully utilize the available computing resources. Since the number of cores in an embedded CPU has increased rapidly, the processing power and the potential for parallelism of embedded CPUs have increased greatly. Also, it has become feasible for tasks that were originally intended to run on application-specific processors to be executed on an embedded CPU with the OpenCL framework. Therefore, the OpenCL framework for embedded multi-core CPUs is needed to fully utilize a heterogeneous embedded system.

However, the study of the OpenCL framework for embedded multi-core CPUs is in a rudimentary stage. Hence, in this paper, a novel OpenCL framework for embedded multi-core CPUs is proposed. How the proposed OpenCL framework is designed, and how various execution environment models are defined for an embedded multi-core CPU will be discussed. To implement the proposed framework, the compilation environment for both the host program and OpenCL kernels has been developed. OpenCL libraries have also been implemented. A performance evaluation was carried out with respect to various definitions on the device architecture and the execution model. When running on embedded multi-core CPUs, applications parallelized by OpenCL C++ showed much better performance than those written in C++. Furthermore, since programmers are capable of managing hardware resources and threads using OpenCL APIs automatically, efficient computing on a heterogeneous computing platform can be easily achieved.

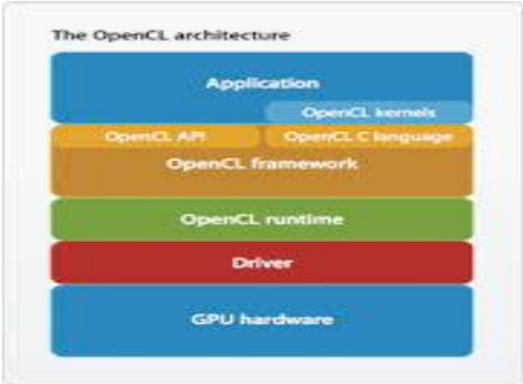

Figure1.1: Architecture of OpenCL

The OpenCL standard open programming standards designers are tasked with a very challenging objective: arrive at a common set of programming standards that are acceptable to a range of competing needs and requirements. The Khronos consortium that manages the OpenCL standard has done a good job addressing these requirements. The consortium has developed an applications programming interface (API) that is general enough to run on significantly different architectures while being adaptable enough that each hardware platform can still obtain high performance. Using the core language and correctly following the specification, any program designed for one vendor can execute on another's hardware. The model set forth by OpenCL creates portable, vendor- and device-independent programs that are capable of being accelerated on many different hardware platforms.

The OpenCL API is a C++ with a CþþWrapper API that is defined in terms of the C API. There are third-party bindings for many languages, including Java, Python, and .NET. The code that executes on an OpenCL device, which in general is not the same device as the host CPU, is written in the OpenCL C++ language. OpenCL C++is a restricted version of the C99 language with extensions appropriate for executing data-parallel code on a variety of heterogeneous devices.

The OpenCL specification is defined in four parts, called models, that can be summarized as follows:

1. Platform model: Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C++ code (the devices).It defines an abstract hardware model that is used by programmers when writing OpenCL C++ functions (called kernels) that execute on the devices.
2. Execution model: Defines how the OpenCL environment is configured on the host and how kernels are executed on the device. This includes setting up an OpenCL context on the host, providing mechanisms for host–device interaction, and defining a concurrency model used for kernel execution on devices.
3. Memory model: Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies, although this has not limited adoptability by other accelerators.
4. Programming model: Defines how the concurrency model is mapped to physical hardware.

Today's computing environments are becoming more multifaceted, exploiting the capabilities of a range of multi-core microprocessors, central processing units (CPUs), digital signal processors, reconfigurable hardware (FPGAs), and graphic processing units (GPUs). Presented with so much heterogeneity, the process of developing efficient software for such a wide array of architectures poses a number of challenges to the programming community. Applications possess a number of workload behaviors, ranging from control intensive (e.g., searching, sorting, and parsing) to data intensive (e.g., image processing, simulation and modeling, and data mining). Applications can also be characterized as compute intensive (e.g., iterative methods, numerical methods, and financial modeling), where the overall throughput of the application is heavily dependent on the computational efficiency of the underlying hardware. Each of these workload classes typically executes most efficiently on a specific style of hardware architecture. No single architecture is best for running all classes of workloads, and most applications possess a mix of the workload characteristics. For instance, control-intensive applications tend to run faster on superscalar CPUs, where significant die real estate has been devoted to branch prediction mechanisms, whereas data-intensive applications tend to run fast on vector architectures, where the same operation is applied to multiple data items concurrently.

II. BLOCK DIAGRAM

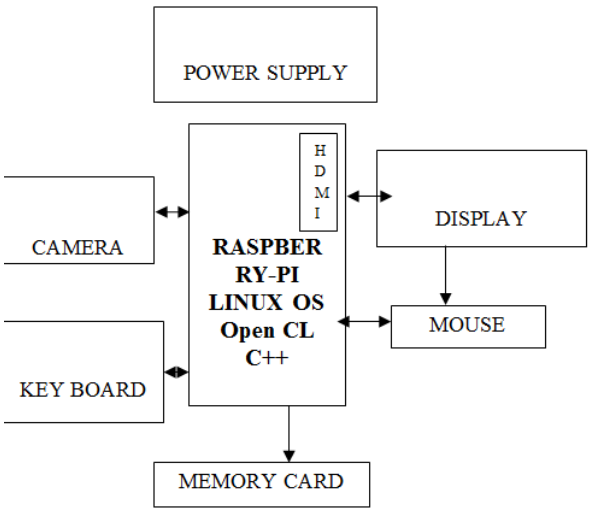

Figure 1.2: Block Diagram

A webcam is a compact digital camera you can hook up to your computer to broadcast video images in real time (as they happen). Just like a digital camera, it captures light through a small lens at the front using a tiny grid of microscopic light-detectors built into an image-sensing microchip (either a charge-coupled device (CCD) or, more likely these days, a CMOS image sensor). As we'll see in a moment, the image sensor and its circuitry converts the picture in front of the camera into digital format—a string of zeros and ones that a computer knows how to handle. Unlike a digital camera, a webcam has no built-in memory chip or flash memory card: it doesn't need to "remember" pictures because it's designed to capture and transmit them immediately to a computer. That's why webcams have USB cables coming out of the back. The USB cable supplies power to the webcam from the computer and takes the digital information captured by the webcam's image sensor back to the computer—from where it travels on to the Internet.

Photo: Unlike the webcam above, which you can focus by twisting its lens, this Microsoft life cam VX-800 has a preset focus. If you look closely, you can just see the power indicator light (top left, not currently lit up) and the microphone (top right). The stand can simply rest on a table or open up to clip on top of your laptop.

How does an image sensor chip work? All webcams work in broadly the same way: they use an image sensor chip to catch moving images and convert them into streams of digits that are uploaded over the Internet. The image sensor chip is the heart of a webcam—so how does that bit work? Let's take a webcam apart and find out. Take the outer case off a webcam and you'll find it's little more than a plastic lens mounted directly onto a tiny electronic circuit board underneath. The lens screws in and out to increase its focal length, controlling the focus of your cam:

Now take the lens off and you can see the image sensor (CCD or CMOS chip): it's the square thing in the middle of this circuit. Only the tiny, green-colored central part is light-sensitive: the rest of the chip is concerned with connecting the light detector to the bigger circuit that surrounds it: Webcams versus digital cameras so the image sensor is the "electronic eye" of a webcam or a digital camera. It's a semiconductor chip made of millions of tiny, light-sensitive squares arranged in a grid pattern. These squares are called pixels.

Basic webcams use relatively small sensors with just a few hundred thousand pixels (typically a grid of 640 × 480. Good digital cameras use sensors with many more pixels; that's why cameras are compared by how many megapixels (millions of pixels) they have. A basic webcam has about 0.3 megapixels (300,000, in other words), while a digital camera with 6 megapixels has over 20 times more—probably arranged in a rectangle with three thousand across and two thousand down (3000 x 2000 = 6 million). A better camera rated at 12 megapixels would have a 4000 x 3000 pixel sensor. Take a photo the same size with those two cameras and the 12 megapixel one is going to give you 1000 more dots horizontally and 1000 more vertically—smaller dots giving more detail and higher resolution. A single pixel in a really good sensor is something like 10 micrometers (10µm) in diameter (5–10 times smaller than the diameter of a typical human hair)!

How does an image sensor convert a picture into digital form? When you take a digital photo or stare into your webcam, light zooms into the lens. This incoming "picture" hits the image sensor, which breaks it up into individual pixels that are converted into numeric form. Ccds and CMOS chips, the two kinds of image sensor, do this job in slightly different ways. Both initially convert incoming light rays into electricity, much like photoelectric cells (used in things like "magic eye" intruder alarms or

restroom washbasins that switch on automatically when you put your hands under the faucet). But a CCD is essentially an analog optical chip that converts light into varying electrical signals, which are then passed on to one or more other chips where they're digitized (turned into numbers). By contrast, a CMOS chip does everything in one place: it captures light rays and turns them into digital signals all on the one chip. So it's essentially a digital device where a CCD is an analog one. CMOS chips work faster and are cheaper to make in high volume than ccds, so they're now used in most low-cost cell phone cameras and webcams. But ccds are still widely used in some applications, such as low-light astronomy.

Whether images are being generated by a CMOS sensor or a CCD and other circuitry, the basic process is the same: an incoming image is converted into an outgoing pattern of digital pixels. Let's just refer to "the image sensor" from now on (and forget about whether it's a CCD and other chips or a CMOS sensor). First, the image sensor measures how much light is arriving at each pixel. This information is turned into a number that can be stored on a memory chip inside the camera. Thus, taking a digital photograph converts the picture you see into a very long string of numbers. Each number describes one pixel in the image—how bright or dark and what color it is.

Step by step

1. Light from the object (in this case, a bicycle) enters the camera lens.
2. The image sensor inside the camera splits the image up into millions of pixels (squares). An LCD display on the back of the camera shows you the image that the sensor is capturing—not an image of the object seen through a series of lenses (as with a conventional camera), but a redrawn, computerized version of the original object displayed on a screen.
3. The sensor measures the color and brightness of each pixel.
4. The color and brightness are stored as binary numbers (patterns of zeros and ones) in the camera's memory card. When you connect your camera to a computer, these numbers are transmitted instantly down the wire.

III. HARDWARE REQUIREMENTS

- Raspberry Pi Development Board
- UVC Driver Camera
- Display

EXTRA HARDWARE:

- SD card containing Linux Operating system
- USB keyboard, Mouse
- TV or monitor (with HDMI, DVI, Composite or SCART input)
- Power supply
- Video cable to suit the TV or monitor used

1.5. SOFTWARE REQUIREMENTS:

- OpenCL C++
- C++
- Linux

IV. METHODOLOGY

4.1. EXISTING SYSTEM:

In existing system Programming languages use the memory which will allocates in ram and CPU registers. While communicating the graphics we need an external graphic card. While communicating with CPU and graphic card some time delay where produced. to avoid the time delay we are allocating memory in graphic card reduces the time taking to multiplying algorithms such as opencv and Matlab function could be easier and faster calculations can be expected. To overcome this problem implementing the proposed method.

4.2. PROPOSED SYSTEM:

In this proposed method, to avoid the time delay. so for this we are approaching an opencl (open computer language). OpenCL was proposed to provide a framework that supports heterogeneous computing platforms. which is having advanced features support to complete task on time . In order to overcome the disadvantage in existing method.In C++ has no security, Complex in very large high level program and C++ not support dynamic memory allocation whereas in OpenCL C++ has rich library which provides a number of built-in functions. It also offers dynamic memory allocation.

In modern mobile embedded systems, various energy-efficient hardware acceleration units are employed in addition to a multi-core CPU. To fully utilize the computational power in such heterogeneous systems, Open Computing Language (OpenCL) has been proposed. A key benefit of OpenCL is that it works on various computing platforms. However, most vendors offer OpenCL software development kits (SDKs) that support their own computing platforms. The study of the OpenCL framework for embedded multi-core CPUs is in a rudimentary stage. In this paper, an OpenCL framework for embedded multi-core CPUs that dynamically redistributes the time-varying workload to CPU cores in real time is proposed. A compilation environment for both host programs and OpenCL kernel programs was developed and OpenCL libraries were implemented. A performance evaluation was carried out with respect to various definitions of the device architecture and the execution model. When running on embedded multi-core CPUs, applications parallelized by OpenCL C++ showed much better performance than the applications written in C++ without parallelization. Furthermore, since programmers are capable of managing hardware resources and threads using OpenCL application programming interfaces (APIs) automatically, highly efficient computing both in terms of the performance and energy consumption on a heterogeneous computing platform can be easily achieved.

V. SCHEMATIC DIAGRAM

ISSN (ONLINE): 2250-0758, ISSN (PRINT): 2394-6962

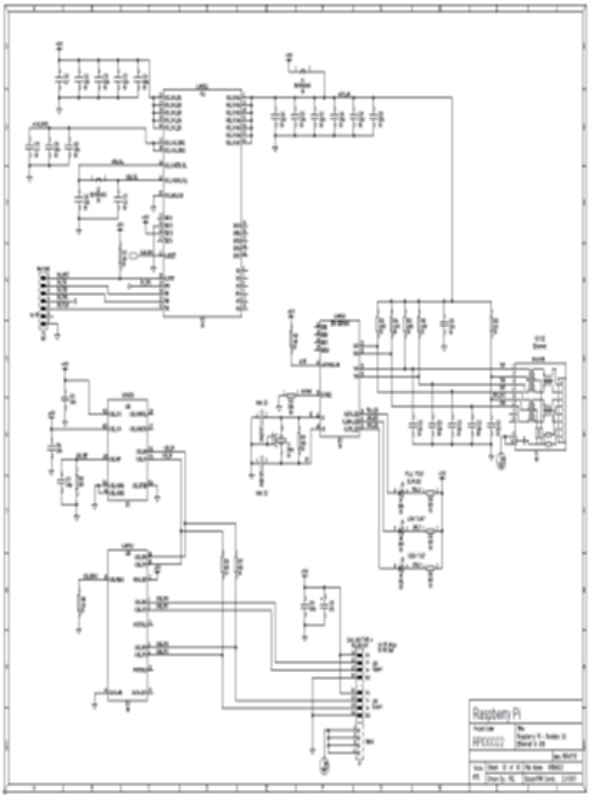

Figure: Schematic Diagram

V. EXPERIMENTAL RESULTS

In this paper, we are giving the complete description on the proposed system architecture. Here we are using Raspberry Pi board as our platform. It has an ARM-11 SOC with integrated peripherals like USB, Ethernet etc. On this board we are installing Linux operating system with necessary drivers for all peripheral devices.

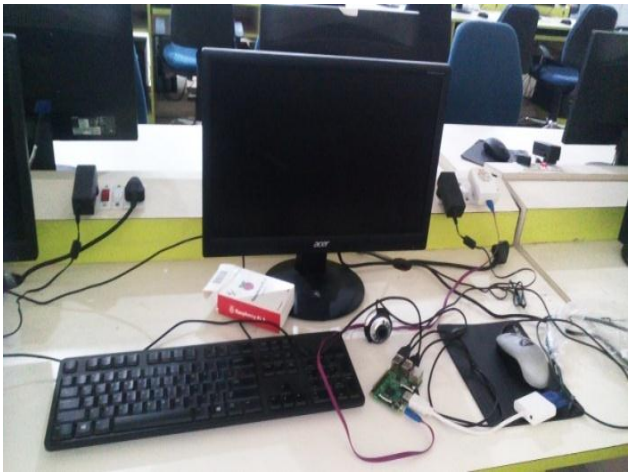

Figure 5.1: Complete project after connections

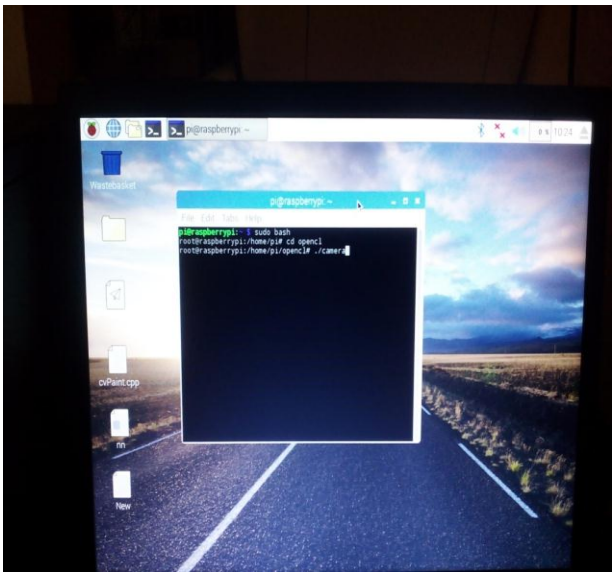

Figure 5.2: Linux command to open C++ video window

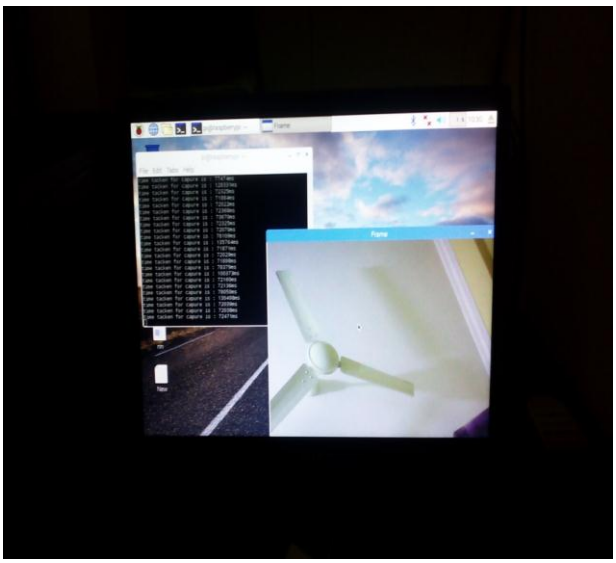

Figure 5.3: C++ output of receiving a video encoding with time delay

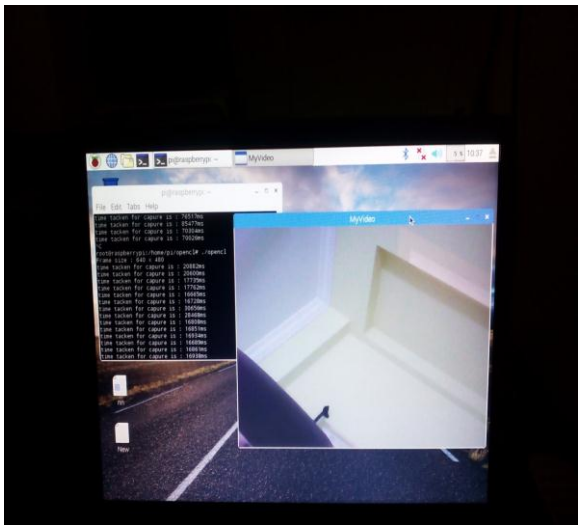

Figure 5.4: Linux command to open OpenCL C++ video window

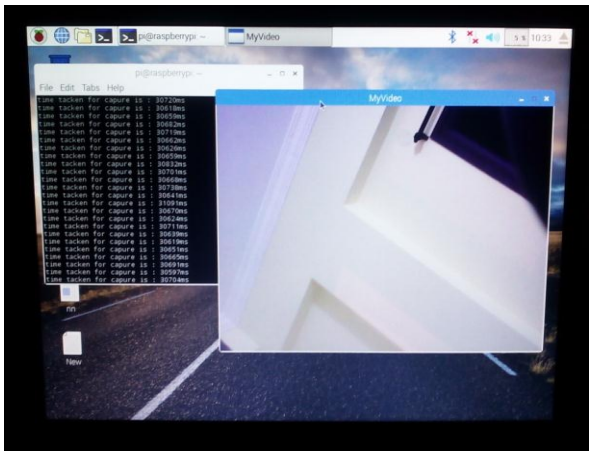

Figure 5.5: OpenCL C++ output of receiving a video encoding without time delay(live show)

DIFFERENCE BETWEEN C++ AND OPENCL C++

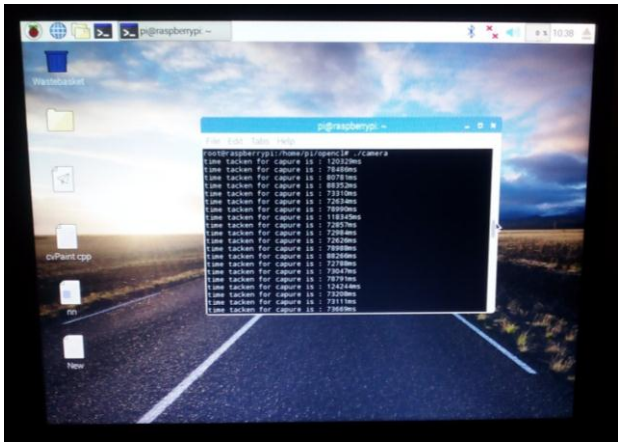

Figure 5.6: In C++ time taken for video capture from 72626ms to 124244ms in one minute time duration

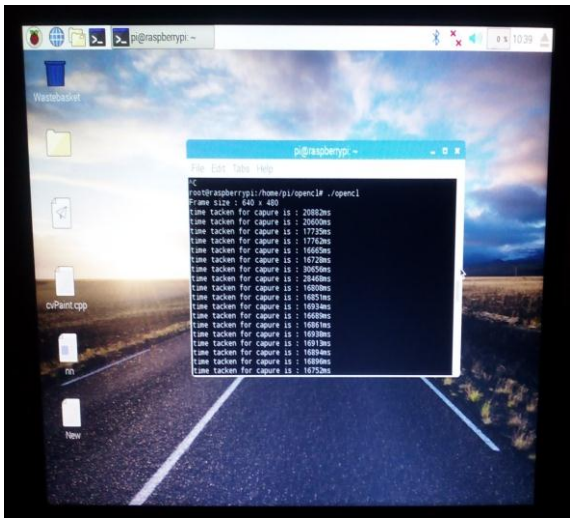

Figure 5.7: In OpenCL C++ time taken for video capture from 16665ms to 30656ms in one minute time duration

The paper **"Design of OpenCL framework for embedded multi-core processors"** has been successfully designed and tested. It has been developed by integrating features of all the hardware components and software used. Presence of every module has been reasoned out and placed carefully thus contributing to the best working of the unit. Secondly, using highly advanced Raspberry pi board and with the help of growing technology the paper has been successfully implemented.

VI. FUTURE SCOPE

Open Graphics Library (**OpenGL**) is a cross-language, cross-platform application program interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three−dimensional applications.

OpenGL is designed as a streamlined, hardware−independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using. Similarly, OpenGL doesn't provide high−level commands for describing models of three−dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of geometric primitives¾points, lines, and polygons.

A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation. Also, there is a higher−level, object−oriented toolkit, Open Inventor, which is built atop OpenGL, and is available separately for many implementations of OpenGL

REFERENCES

[1] http://www.khronos.org/opencl/
[2] http://developer.amd.com/gpu/ATIstreamSDK/pages/publications.aspx
[3] elinux.org/Processors
[4] www.raspberrypi.org