GPGPU WITH GLES

General Purpose Computations on Graphical Processing Unit of BeagleBone Black using OpenGL ES

Project Report

Eklavya 2022

Society of Robotics and Automation, Veermata Jijabai Technological Institute, Mumbai

August 2022

ACKNOWLEDGEMENT

A project of such magnitude is realized in erudite moments. In this regard, we are extremely grateful to our mentors, Krishna Narayanan and Kunal Agarwal, for their unflinching support and encouragement throughout this journey. We also extend our gratitude and appreciation to SRA, VJTI for organizing Eklavya and providing us with this wonderful opportunity.

Our Team:

Pratham Deshmukh
mast.deshmukh@gmail.com

Komal Sambhus
komalss3210@gmail.com

Contents

1. GPU(Graphics Processing Unit)

1.1. What is a GPU?

A Graphics Processing Unit is an electronic circuit designed to manipulate and alter memory to accelerate the creation of images and graphics for rendering to display devices.

1.2. How is it different from the CPU?

It is somewhat similar to the CPU in architecture but the two differ in the manner in which they execute commands and thus, the number of cores present in them.
Most CPUs have 4 to 8 cores, with each core processing two threads or tasks. A GPU, on the other hand, can have 4 to 10 threads per core. It is capable of quickly rendering images on the screen due to its parallel processing architecture.

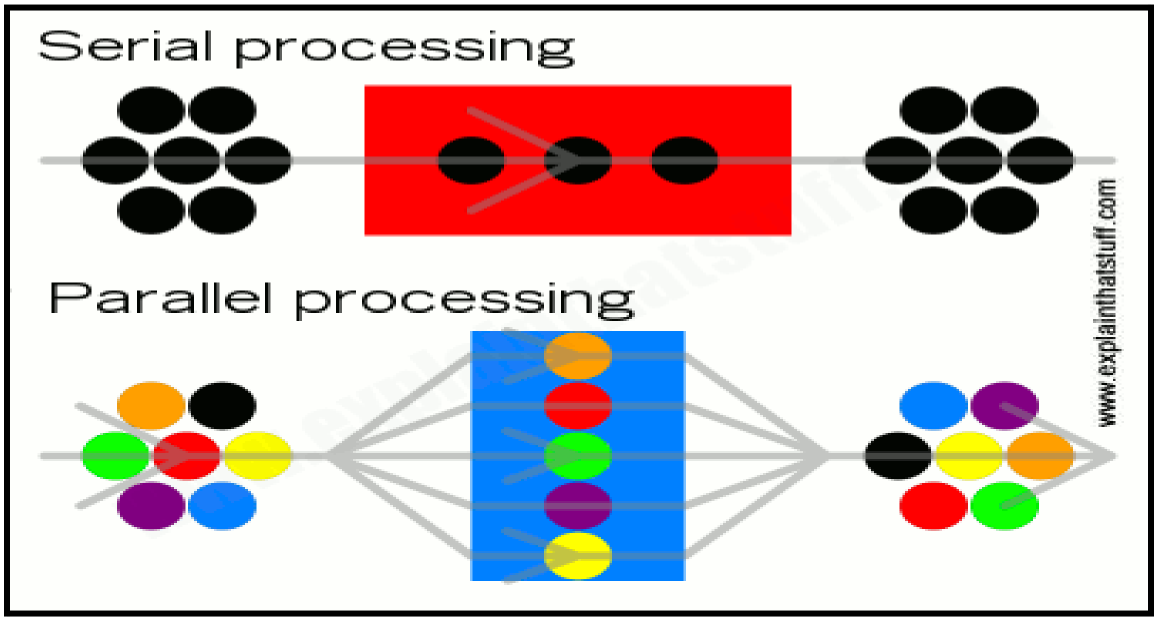
The CPU sends information regarding what needs to appear on screen to the graphics card. In turn, the graphics card takes those instructions and runs them through its own processing unit, rapidly updating its onboard memory (known as VRAM) as to which pixels on the screen need changing.

This information is then transferred from the graphics card to the monitor, where the images, lines, textures, lighting, shading, and everything else changes.

1.3. How does a GPU work?

Video games and other graphic rendering operations require a lot more processing power than the usual programs as every pixel on the screen needs to be computed and in 3D games, geometries and perspectives need to be computed as well. Here, instead of having a couple of big and powerful microprocessors, it is smarter to have a number of tiny microprocessors running at the same time. This is what the GPU is. A very large number of rendering operations are divided among a comparable number of microprocessors which hastens the rendering process.

Another advantage of the GPU is the accelerated computing of math functions via hardware, hence, complicated math functions are resolved directly by the microchips instead of the software.



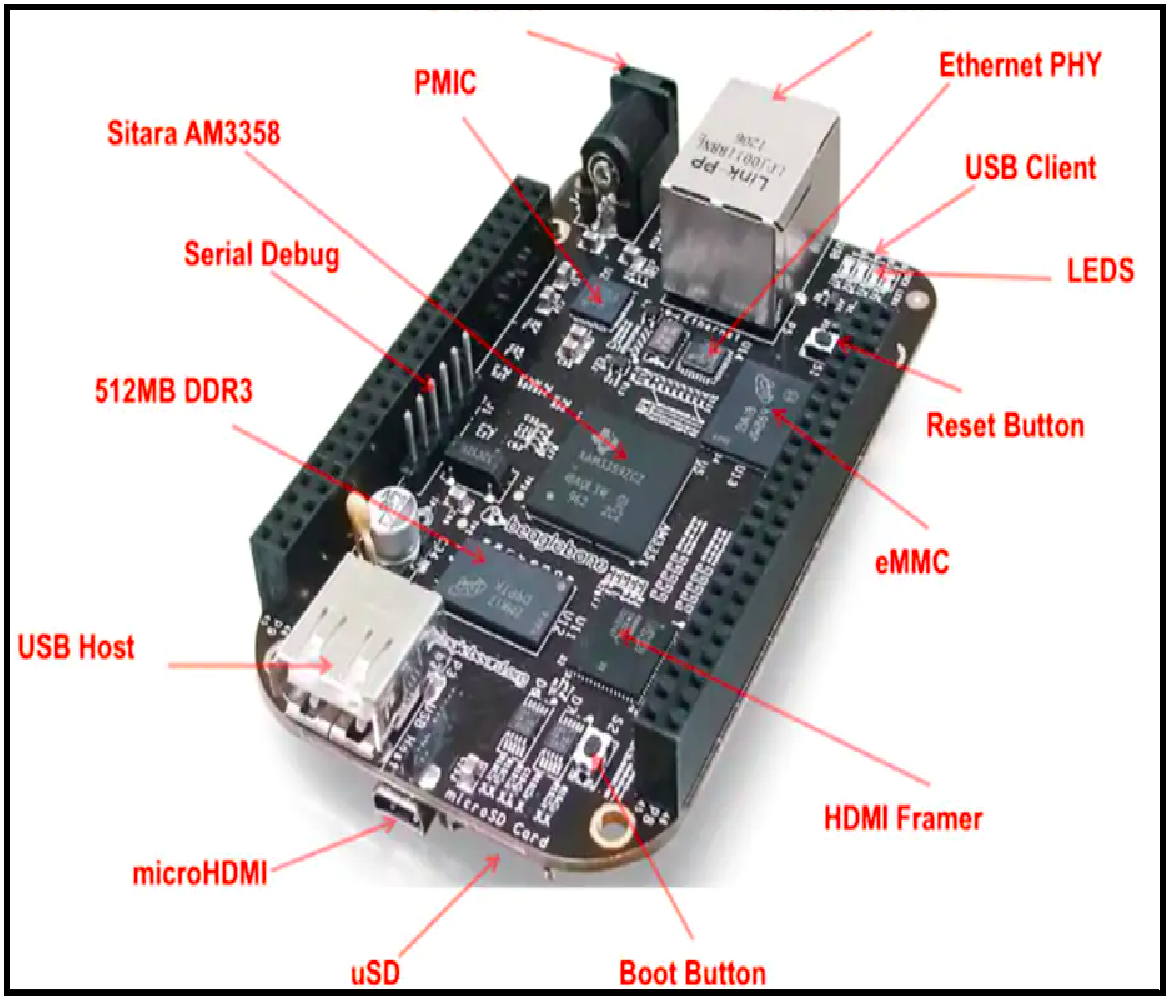
2. BeagleBone Black

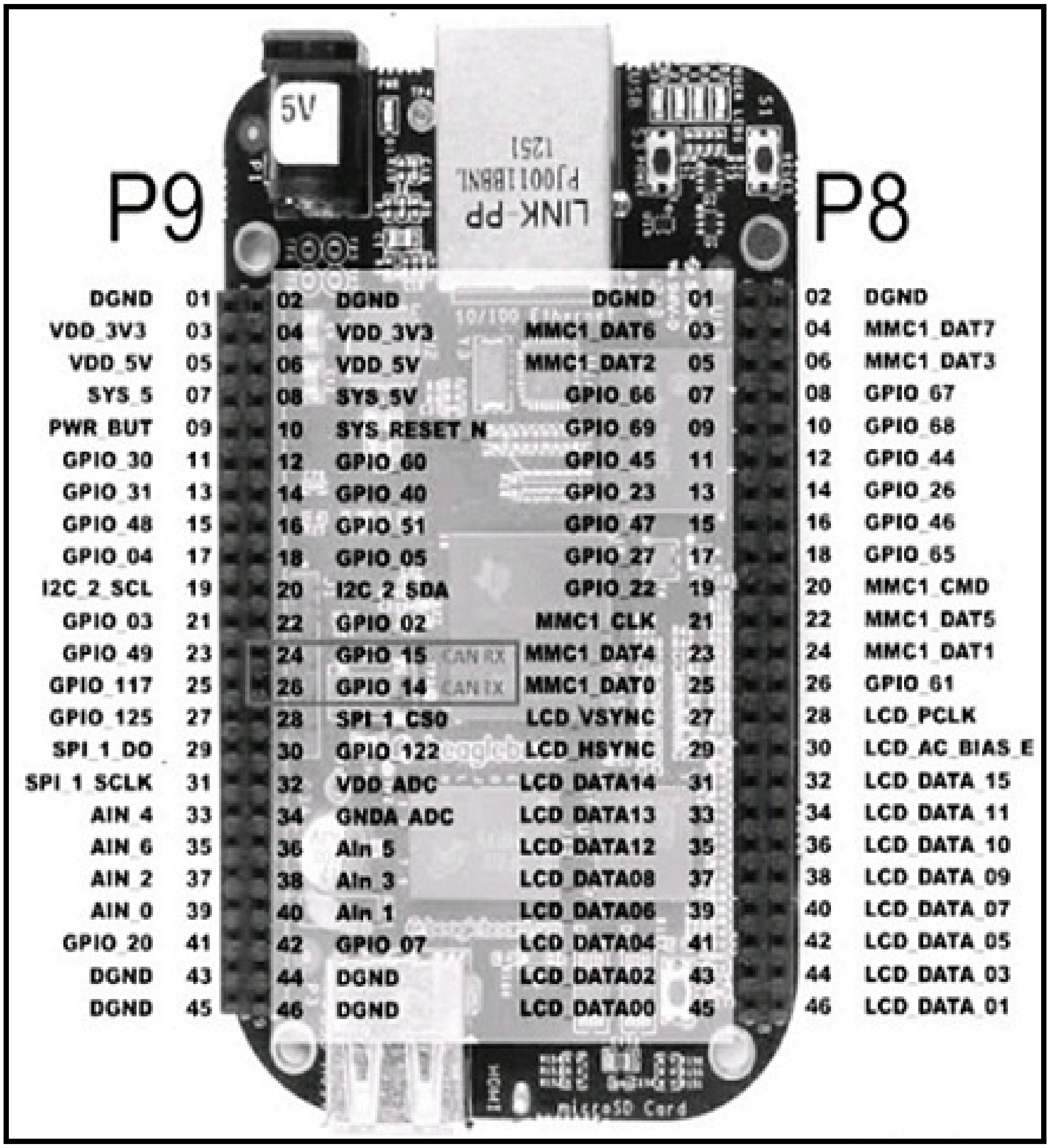
2.1 What is BeagleBone Black?

1.The Beaglebone Black(BBB) is just like a Computer, which comes in a compact package with a processor, graphic acceleration, memory, and all the required ICs soldered to form a single circuit board.

2. Hence, it is also referred to as a Single-Board Computer. It uses a powerful processor called ARM Cortex-A8 processor with 1GHz AM335x.

3. This Beaglebone Black Microcontroller board will provide all the necessary connections for the display, Ethernet network, Mouse, and Keyboard. The booting of this processor is done by using Linux OS.


PMIC
Ethernet PHY
Sitara AM3358
USB Client
Serial Debug
LEDS
512MB DDR3
Reset Button
eMMC
USB Host
HDMI Framer
microHDMI
uSD
Boot Button


P9
P8
5V
LINK-PP
DGND 01 02 DGND
VDD_3V3 03 04 VDD_3V3
VDD_5V 05 06 VDD_5V
SYS_5 07 08 SYS_5V
PWR_BUT 09 10 SYS_RESET_N
GPIO_30 11 12 GPIO_60
GPIO_31 13 14 GPIO_40
GPIO_48 15 16 GPIO_51
GPIO_04 17 18 GPIO_05
I2C_2_SCL 19 20 I2C_2_SDA
GPIO_03 21 22 GPIO_02
GPIO_49 23 24 GPIO_15 CAN RX
GPIO_117 25 26 GPIO_14 CAN TX
GPIO_125 27 28 SPI_1_CS0
SPI_1_DO 29 30 GPIO_122
SPI_1_SCLK 31 32 VDD_ADC
AIN_4 33 34 GNDA_ADC
AIN_6 35 36 AIn_5
AIN_2 37 38 AIn_3
AIN_0 39 40 AIn_1
GPIO_20 41 42 GPIO_07
DGND 43 44 DGND
DGND 45 46 DGND
DGND 01 02 DGND
MMC1_DAT6 03 04 MMC1_DAT7
MMC1_DAT2 05 06 MMC1_DAT3
GPIO_66 07 08 GPIO_67
GPIO_69 09 10 GPIO_68
GPIO_45 11 12 GPIO_44
GPIO_23 13 14 GPIO_26
GPIO_47 15 16 GPIO_46
GPIO_27 17 18 GPIO_65
GPIO_22 19 20 MMC1_CMD
MMC1_CLK 21 22 MMC1_DAT5
MMC1_DAT4 23 24 MMC1_DAT1
MMC1_DAT0 25 26 GPIO_61
LCD_VSYNC 27 28 LCD_PCLK
LCD_HSYNC 29 30 LCD_AC_BIAS_E
LCD_DATA14 31 32 LCD_DATA_15
LCD_DATA13 33 34 LCD_DATA_11
LCD_DATA12 35 36 LCD_DATA_10
LCD_DATA08 37 38 LCD_DATA_09
LCD_DATA06 39 40 LCD_DATA_07
LCD_DATA04 41 42 LCD_DATA_05
LCD_DATA02 43 44 LCD_DATA_03
LCD_DATA00 45 46 LCD_DATA_01

2.2. BeagleBone Black Specifications.

The processor type – Sitara AM3358BZCZ100 with 1 GHz and 2000 MIPS

- Graphics Engine- 20M Polygons/S, SGX530 3D
- Size of SDRAM memory – 512 MB DDR3L, 800 MHz
- Onboard Flash- 8-bit Embedded MMC with 4 GB
- Debug Support – Serial Header, onboard optional 20-pin CTI
- Power Source – mini USB, USB or DC jack; 5 Volts external DC through expansion header
- Type of indicators – 1 power, 2 Ethernet, 4 LEDs, which are user-controllable
- HS USB 2.0 Host Port – Accessible to USB1, Type A Socket, 500 mA LS/FS/HS
- Serial Port – UART0 access via 6-pin 3.3 Volts TTL header. Populated header
- Ethernet – 10/100, RJ45
- User Input – Power button, Reset button, Boot button
- SD/MMC Connector – microSD, 3.3 Volts
- Video out – 16b HDMI, 1280×1024 (max), 1025×768, 1280×720, 1440×900, w/EDID support

- HS USB 2.0 Client Port – Access to USB0, client mode through miniUSB
- Audio – Stereo, via HDMI interface
- Weight – 39.68 gms (1.4 oz)
- Expansion Connectors – 5 Volts, 3.3 Volts power,

2.3. BeagleBone Black Applications

- To study how various graphical processes are performed on BeagleBone Black,
- Motor Controllers,
- It can work as a server in various IoT projects,
- In several projects related to bluetooth connectivity, and
- Used as a signal control unit in several industrial systems.

2.4. Sharing internet connection from linux laptop to BBB using USB.

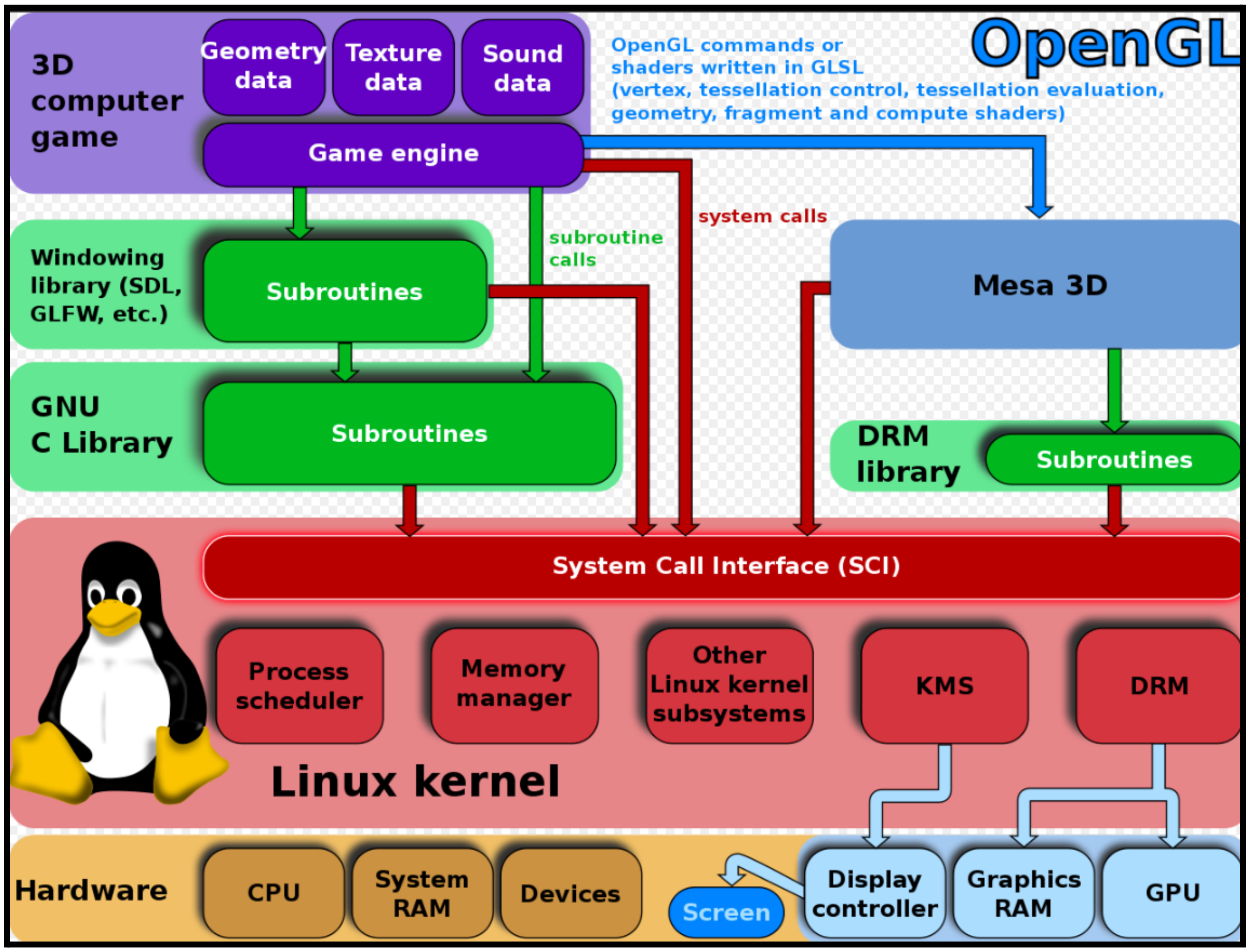
For Installations and to clone and build files on Beaglebone, we will need to connect to it. This can be done as shown here:

https://gist.github.com/pdp7/d2711b5ff1fbb000240bd8337b859412

3. OpenGL

3.1. What is OpenGL?

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



What is an API?

The API (Application Programming Interface) is defined as a set of functions which may be called by the client program, alongside a set of named integer constants (for example, the constant GL_TEXTURE_2D, which corresponds to the decimal.

3.2. Libraries In OpenGL

3.2.1. GLUT Library

The OpenGL Utility Toolkit Library provides high level utilities to simplify OpenGL programming, especially in interacting with the Operating System. It requires very few routines to render graphics using OpenGL. Its routines also take fewer parameters and no pointers are returned. GLUT is designed for simple to moderately complex programs focused on OpenGL rendering. GLUT implements its own event loop. For this reason, mixing GLUT with other APIs that demand their own event handling structure may be difficult.

Following are some functions in a program that includes the GLUT library:

void glutinit(int *argc, char** argv): used to initialize the GLUT library.

glutInitDisplayMode(unsigned int mode): sets the initial display mode.

glutInitWindowSize(int width, int height): sets the initial window size.

glutInitWindowPosition(int x, int y): sets the initial window position.

glutCreateWindow(char **name): creates a top level window and takes in the name of the window as the argument.

glutMainLoop(): enters the GLUT event processing loop.

glutDisplayFunc(void (*func)(void)): sets the display callback to the current window.

3.2.2. GLFW Library

Graphics Library Framework (GLFW) allows users to create and manage OpenGL windows, while handling keyboard, mouse and joystick inputs. GLFW and FreeGLUT are alternatives to the same functions.

glfwInit(): Initializes the GLFW library. It returns GL_TRUE if successful, otherwise GLFW_FALSE.

glfwWindowHint(int target, int hint): This function sets the hints for the next call to glfwCreateWindow() function. The hints, once set, retain their values, until the next call to this function.

glfwCreateWindow(int width, int height, const char* title, GLFWmonitor* monitor, GLFWwindow* share): It creates a window object and its context. This function returns a pointer to this window object.

3.2.3. EGL Library

EGL (Embedded System Graphics Library) is the interface between OpenGL ES and the underlying native display platform. Thus, it is necessary to use OpenGL. It is used to manage the different display buffers and the OpenGL ES context. This library makes it possible for

OpenGL to remain platform independent as it interacts directly with the hardware without having the OpenGL to do it.

This library has been used extensively in this project.

3.2.4. GLEW AND GLAD Libraries

Since OpenGL is merely a specification, it is up to the driver manufacturer to implement the specification to a driver that the specific graphics card supports. OpenGL has many versions and the location of all of its functions is not known at compile-time and needs to be queried at the run-time. This process is OS-specific. To simplify this, the GLAD library is used.

GLEW (OpenGL Extension Wrangler Library) is a cross-platform C/C++ extension loading library that provides an efficient mechanism to determine which extensions are supported on the platform.

These libraries are pretty similar in their functions and either choice does not matter for low-level rendering operations. The choice becomes more prominent for advanced use cases.

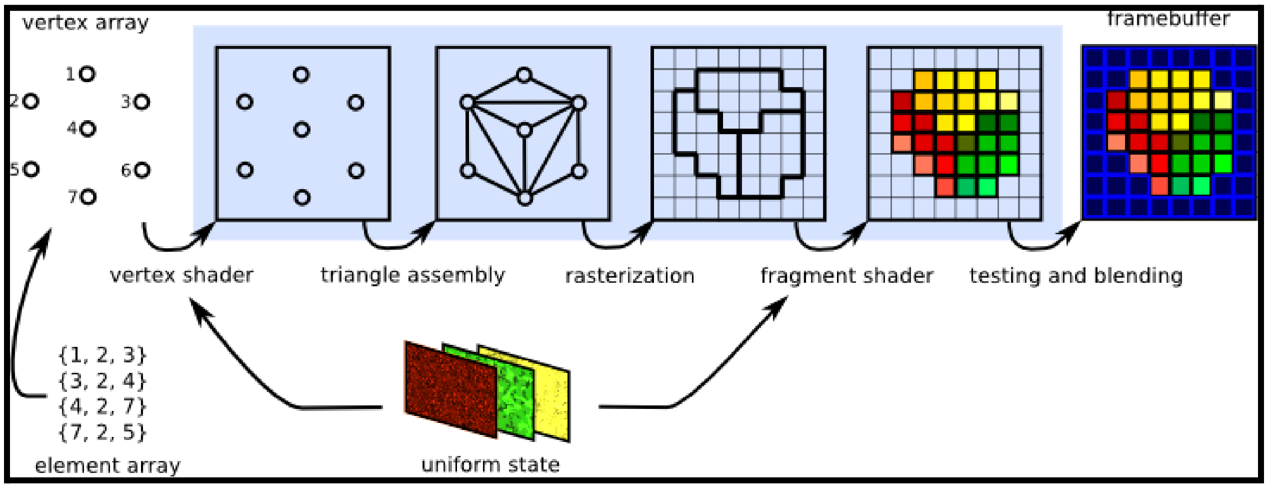
GLAD allows the user to include only those extensions which they wish to, leading to faster compile times. GLEW can detect which dependencies are available at compile time, leading to better adaptability.

3.2.5. GLES 2.0 Library

This library has been used in this repository

All of these libraries are used to interact with the operating system to create a window and display graphics on it. The actual interaction with the GPU is done by the Shaders.

4. Graphics Pipeline



A Graphics Pipeline represents the sequence of steps to create a 2D raster representation of a 3D scene. It shows how the GPU conveys the 3D scene into the computer monitor and can be divided into two large parts: The first transforms the 3D coordinates into 2D coordinates and the second transforms the 2D coordinates into actual colored pixels. This process is divided into several steps where each step requires the output of the previous step as its input. It occurs in a multithreading environment where a lot of code is executed in parallel.

Steps of generating a picture on the screen by the GPU:

A list of 3D coordinates is passed as input to the graphics pipeline. This data is passed to the vertex shader that converts it into points that can be joined to form triangles. The vertex's data is represented as attributes.

The vertex shader takes as input the attributes of a single vertex and does some basic processing on them.

The next step of primitive processing is mostly ignored as its next step creates the actual triangles. It takes as input all the vertices from the vertex shader that form a primitive. It results in a pre-calculation and assembly of all points into triangles, lines and polygons.

The output of the primitive assembly stage is passed to the geometry shader that generates other shapes by emitting new vertices to form other shapes.

This output is passed on to the rasterization stage where the triangles are connected and filled in. The 3D vertices and calculations get converted into a 2D raster. Now, we have a set of pixels from the initial data of vertices.

Fragment shader will give the rasterized pixels texture, color and speculars. It takes individual pixels from the triangle and sends them to the processor. In the end, pixels with colors or textures will be generated.

4.1 Shaders

4.1.1. What are Shaders in OpenGL?

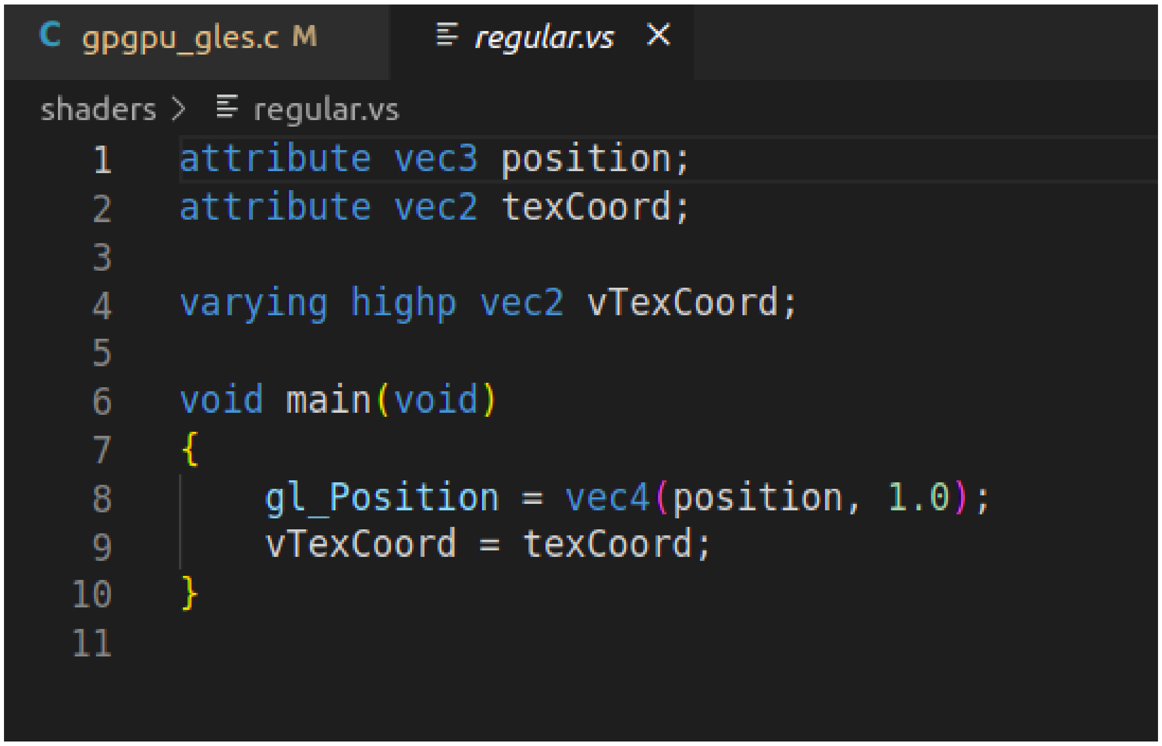
Shaders are small programs that run on the Graphics Processing Unit (GPU) usually found on the graphics card, mainly to manipulate an image before it is drawn on the screen. The term "shader" was originally used to refer only to the fragment (pixel) shaders, but soon enough other uses of shaders were introduced such as the vertex and fragment shaders, making the term more general.

4.1.2. Types of Shaders:

The OpenGL shaders are of different types as defined below:

<u>Vertex Shader:</u>

They modify vertex position, color and texture coordinates. They are defined in the main program by the enum (GL_VERTEX_SHADER).



```
attribute vec3 position;
attribute vec2 texCoord;

varying highp vec2 vTexCoord;

void main(void)
{
    gl_Position = vec4(position, 1.0);
    vTexCoord = texCoord;
}

```

Geometry Shader:

It is the newest of the shader types. It can modify vertices, i.e, procedurally generated geometry. It allows for dynamic /procedural content generation. It is included by the pre-defined enum (GL_GEOMETRY_SHADER).

Fragment Shader (Pixel Shader):

It calculates the colors once for each pixel. The enum for this shader type is GL_FRAGMENT_SHADER.

4.1.3. How are Shaders written?

The three major shoulder languages that have been developed are: GLSL, HLSL and DirectX. Out of these, we have used GLSL in our project.

GLSL is a C-like language with special functions and data types for performing operations on matrices and vectors. Though it is based on C, it adds many more features and functionalities with it. It supports most of the functions of the C language, however, the if structures are not handled well as the GPU is designed to run the same code in exactly the same way in parallel on multiple shader processors. Hence, it has multiple functions such as ‘step’ and ‘clamp’ that give exactly the same results.

Section 2: Project Analysis

1. Project Outputs

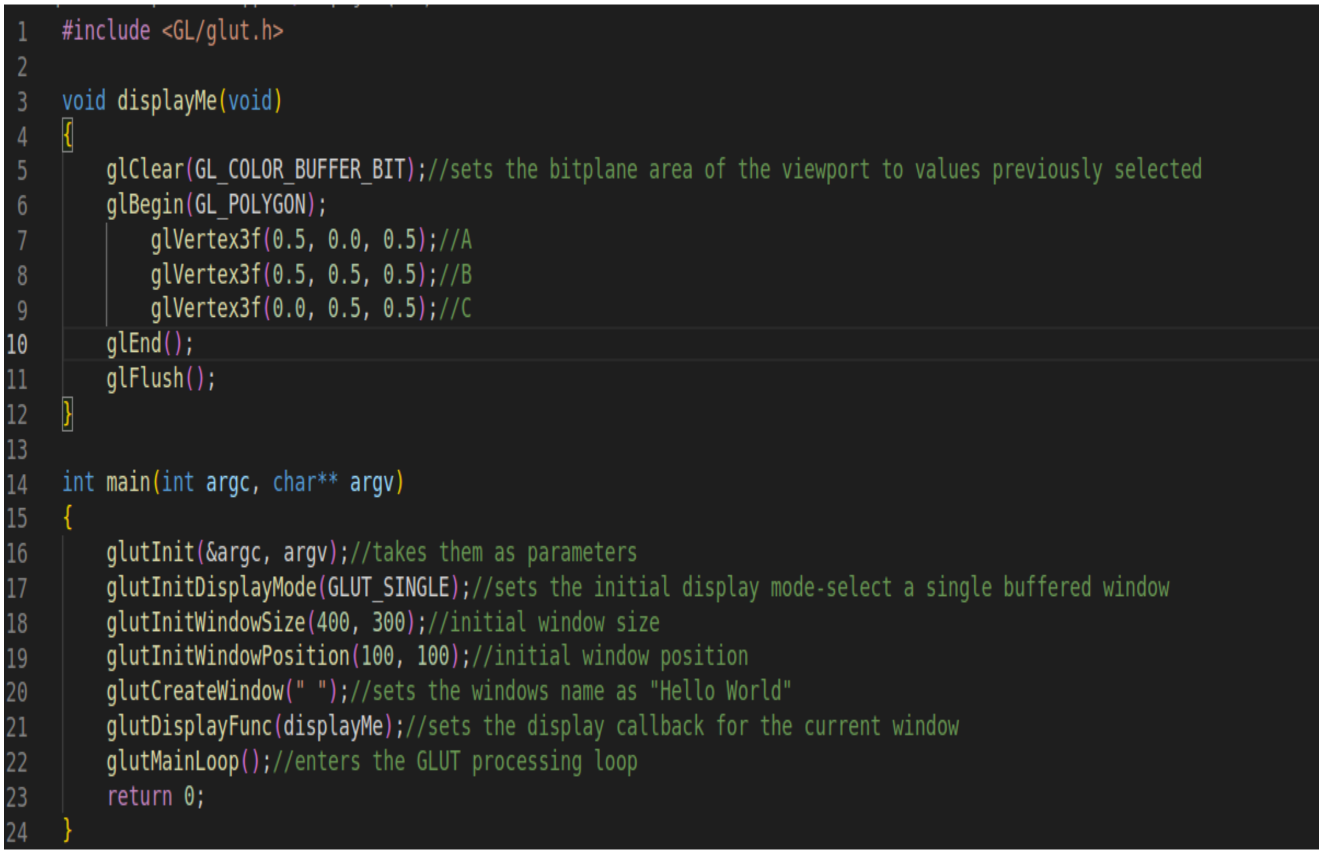
1.1. Triangle and rectangle display using GLUT library

As stated earlier, GLUT is the OpenGL Utility Toolkit which implements a simple windowing application programming interface for OpenGL.

We have simply programmed a code using GLUT to display Triangle and rectangle

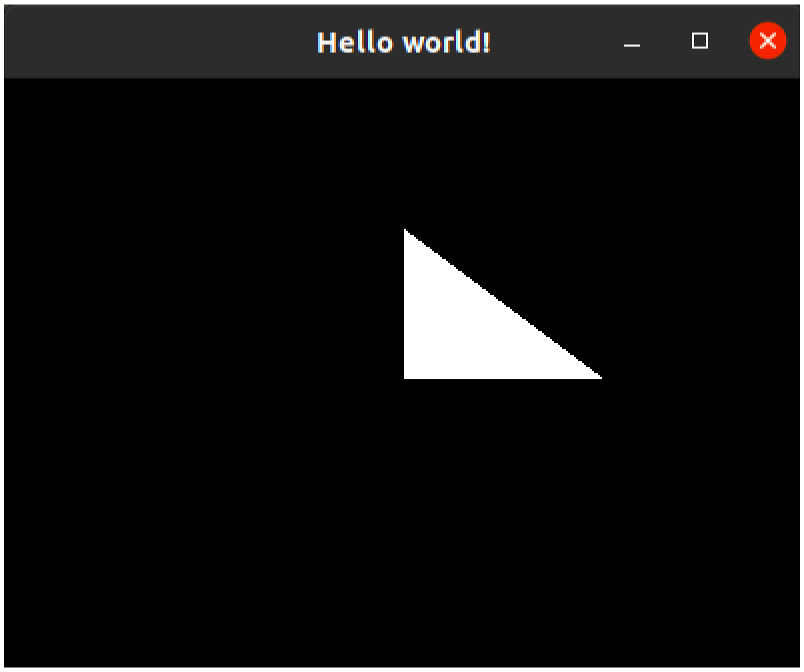
Triangle:

Code:



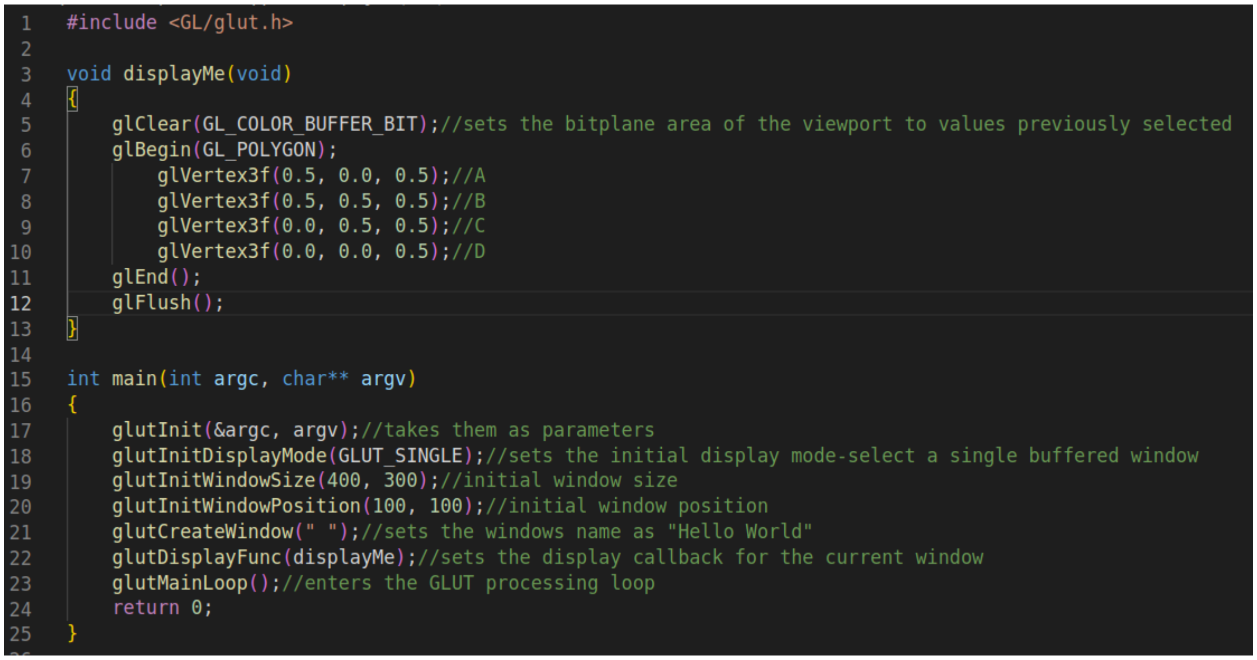
```
#include <GL/glut.h>

void displayMe(void)
{
    glClear(GL_COLOR_BUFFER_BIT);//sets the bitplane area of the viewport to values previously selected
    glBegin(GL_POLYGON);
        glVertex3f(0.5, 0.0, 0.5);//A
        glVertex3f(0.5, 0.5, 0.5);//B
        glVertex3f(0.0, 0.5, 0.5);//C
    glEnd();
    glFlush();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);//takes them as parameters
    glutInitDisplayMode(GLUT_SINGLE);//sets the initial display mode-select a single buffered window
    glutInitWindowSize(400, 300);//initial window size
    glutInitWindowPosition(100, 100);//initial window position
    glutCreateWindow(" ");//sets the windows name as "Hello World"
    glutDisplayFunc(displayMe);//sets the display callback for the current window
    glutMainLoop();//enters the GLUT processing loop
    return 0;
}
```

Output:



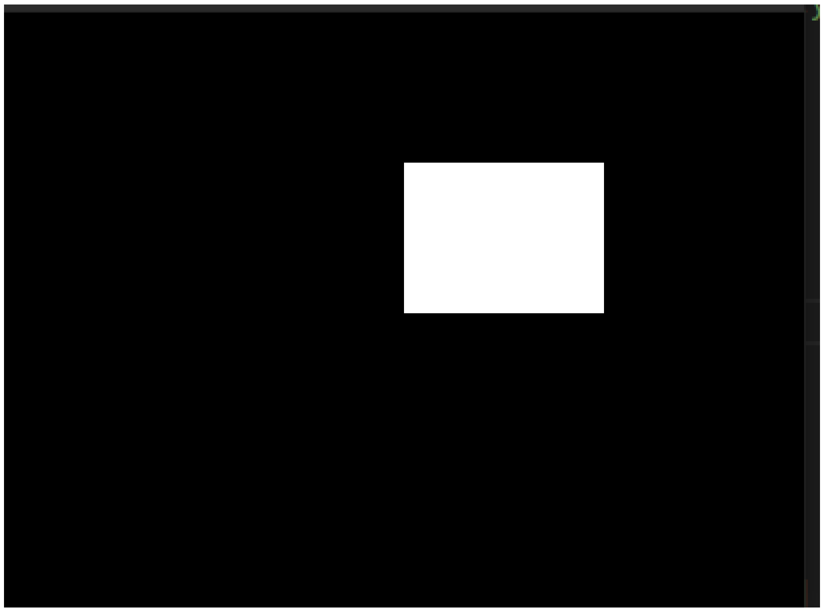
Rectangle:

Code:



```
#include <GL/glut.h>

void displayMe(void)
{
    glClear(GL_COLOR_BUFFER_BIT);//sets the bitplane area of the viewport to values previously selected
    glBegin(GL_POLYGON);
        glVertex3f(0.5, 0.0, 0.5);//A
        glVertex3f(0.5, 0.5, 0.5);//B
        glVertex3f(0.0, 0.5, 0.5);//C
        glVertex3f(0.0, 0.0, 0.5);//D
    glEnd();
    glFlush();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);//takes them as parameters
    glutInitDisplayMode(GLUT_SINGLE);//sets the initial display mode-select a single buffered window
    glutInitWindowSize(400, 300);//initial window size
    glutInitWindowPosition(100, 100);//initial window position
    glutCreateWindow(" ");//sets the windows name as "Hello World"
    glutDisplayFunc(displayMe);//sets the display callback for the current window
    glutMainLoop();//enters the GLUT processing loop
    return 0;
}
```

Output:



1.2. Analysis of Shaders

The Vertex Shader

The vertex shader consists of the attributes ‘position’ and ‘vTexCoord’ which are respectively the inputs for the three dimensional position and the texture coordinates of the vertex. The outputs are represented by the defined high precision vec2 ‘vTexCoord’, and the in-built variable gl_Position that returns a vec4.

1.3. Matrix Multiplication using GLES

This was one of the original goals of our project. Using the code for Array Addition as a reference, we coded for matrix multiplication. The changes for the algorithm were made in the fragment shader file as that is where the actual computation takes place.

Samplers ‘texture0’ and ‘texture1’ bring in the 2 matrices which are initialized in the mat_mult_int.c file. The ‘varying vec2 vTexCoord’ represents the coordinates of the pixel for which we have to calculate the product.

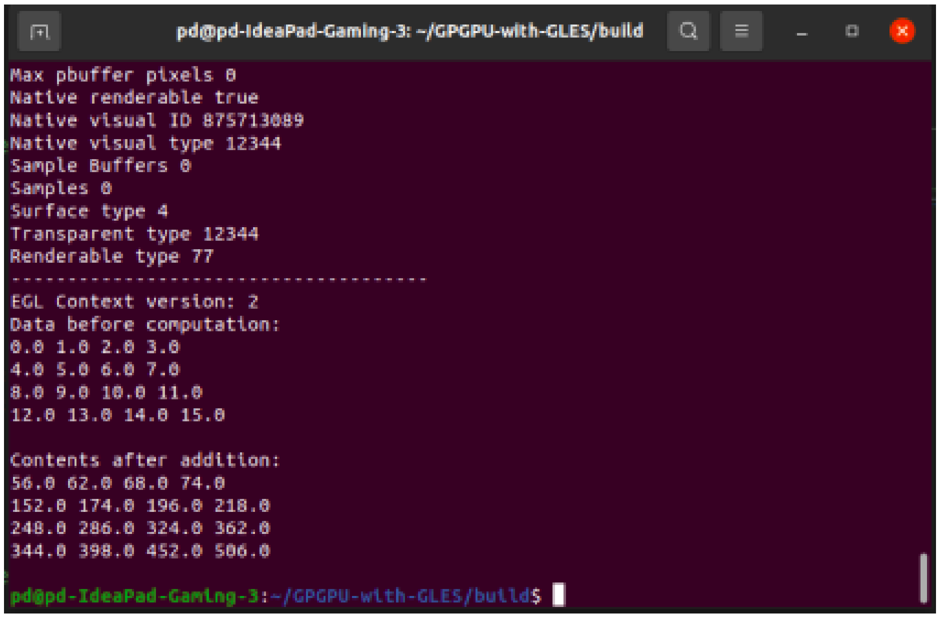
The fragment shader runs for each pixel on the screen, hence, we write the algorithm targeting multiplication between the row of the first matrix with the corresponding column of the second matrix. This ensures that the calculations lead to a value corresponding to only one pixel. The input textures (matrices) are converted into floating point numbers. This is done using: unpack(texture2D(texture0, vec2(i, k/4.0))); A similar function is used for the second matrix.

The vec2 coordinates are used to access the individual elements of the two matrices. ‘texture0’ specifies the texture we are working on and the texture2D function returns a vec4 color value of the pixel. This is converted into a floating point number using the unpack function. The calculations are done using the floating point numbers as it becomes difficult to work with vectors.

The floating point result in the shader is converted to the vec4 format using the pack function. Once this computation has been completed, the result is stored in the in-built variable of the vec4 format gl_FragColor that is supposed to return the color of the pixel. This vec4 is then returned to the CPU to be printed on the screen.

The Shaders code:
https://github.com/Pratham-Bot/GPGPU-with-GLES/blob/main/shaders/mult_mat_int.fs

Output:



```
Max pbuffer pixels 0
Native renderable true
Native visual ID 875713089
Native visual type 12344
Sample Buffers 0
Samples 0
Surface type 4
Transparent type 12344
Renderable type 77
-------------------------------------
EGL Context version: 2
Data before computation:
0.0 1.0 2.0 3.0
4.0 5.0 6.0 7.0
8.0 9.0 10.0 11.0
12.0 13.0 14.0 15.0

Contents after addition:
56.0 62.0 68.0 74.0
152.0 174.0 196.0 218.0
248.0 286.0 324.0 362.0
344.0 398.0 452.0 506.0

pd@pd-IdeaPad-Gaming-3:~/GPGPU-with-GLES/build$
```

1.4. Multiplication of Array by a Scalar

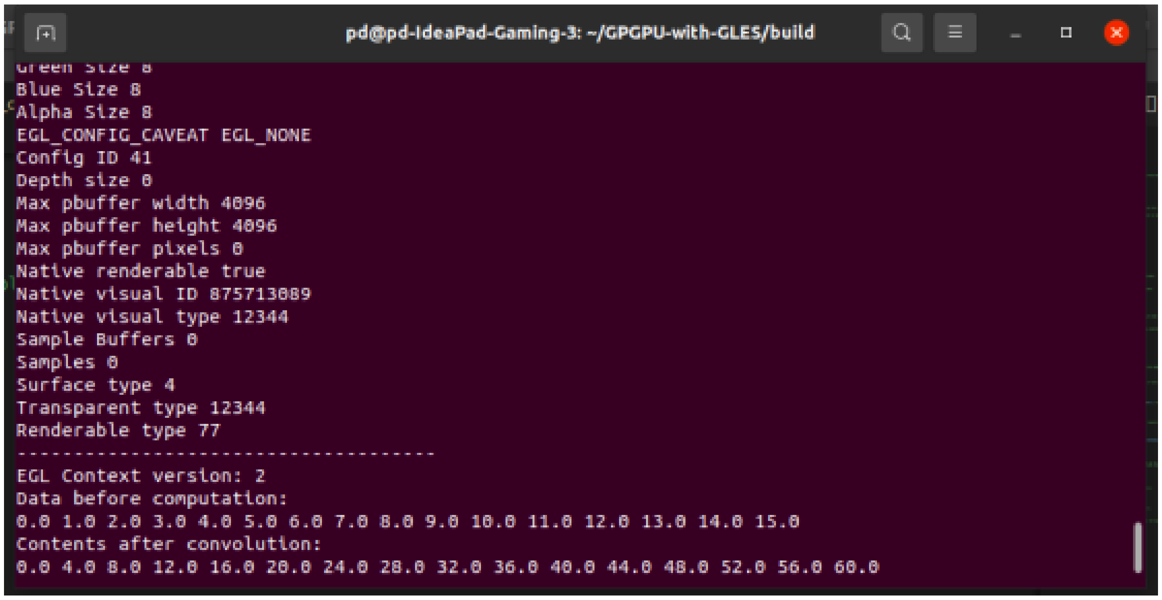
This code was referenced from the 2D convolution algorithm. The inputs and outputs remain the same as in Matrix Multiplication.

The 'if' conditions at the beginning of the void main function make sure that the operation is performed on elements present inside the input matrix. We then load the input matrix into a sample.Its indices need to be changed in only one direction. When the matrix is fully loaded, it is multiplied with the kernel and the result of each pixel is returned via the gl_FragColor.

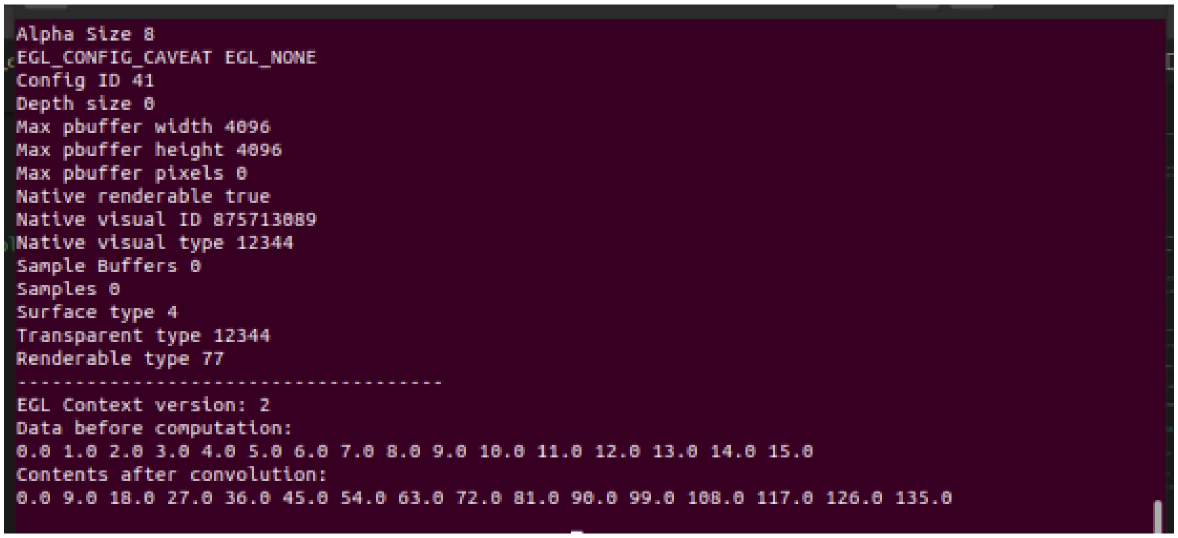
Shader Code:

https://github.com/Pratham-Bot/GPGPU-with-GLES/blob/main/shaders/array_x4.fs https://github.com/Pratham-Bot/GPGPU-with-GLES/blob/main/shaders/array_x9.fs

Output:



```
pd@pd-IdeaPad-Gaming-3: ~/GPGPU-with-GLES/build
Green Size 8
Blue Size 8
Alpha Size 8
EGL_CONFIG_CAVEAT EGL_NONE
Config ID 41
Depth size 0
Max pbuffer width 4096
Max pbuffer height 4096
Max pbuffer pixels 0
Native renderable true
Native visual ID 875713089
Native visual type 12344
Sample Buffers 0
Samples 0
Surface type 4
Transparent type 12344
Renderable type 77
-------------------------------------
EGL Context version: 2
Data before computation:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0
Contents after convolution:
0.0 4.0 8.0 12.0 16.0 20.0 24.0 28.0 32.0 36.0 40.0 44.0 48.0 52.0 56.0 60.0
```



```
Alpha Size 8
EGL_CONFIG_CAVEAT EGL_NONE
Config ID 41
Depth size 0
Max pbuffer width 4096
Max pbuffer height 4096
Max pbuffer pixels 0
Native renderable true
Native visual ID 875713089
Native visual type 12344
Sample Buffers 0
Samples 0
Surface type 4
Transparent type 12344
Renderable type 77
-------------------------------------
EGL Context version: 2
Data before computation:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0 13.0 14.0 15.0
Contents after convolution:
0.0 9.0 18.0 27.0 36.0 45.0 54.0 63.0 72.0 81.0 90.0 99.0 108.0 117.0 126.0 135.0
```

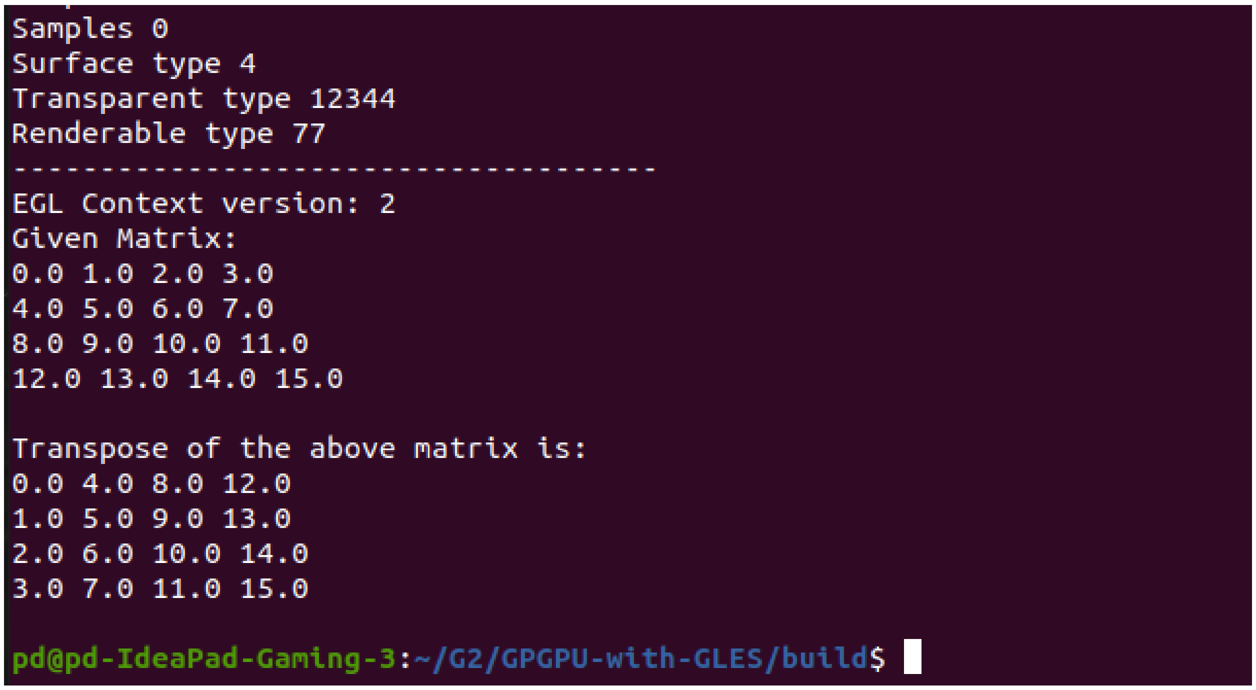
1.5. Transpose of a Matrix

The transpose of the input matrix was computed using the simple logic of inversing the components of the input coordinates and returning the value at the new texture coordinates. The coordinates are split into variables i and j, and the value of the vector at (j,i) is returned via the gl_FragColor variable.Thus, at the end of the computation, the transpose of the entire input matrix is returned.

Shader Code:

https://github.com/Pratham-Bot/GPGPU-with-GLES/blob/Komal-Dev/shaders/matrix_transpose.fs

Output:



```
Samples 0
Surface type 4
Transparent type 12344
Renderable type 77
-------------------------------------
EGL Context version: 2
Given Matrix:
0.0 1.0 2.0 3.0
4.0 5.0 6.0 7.0
8.0 9.0 10.0 11.0
12.0 13.0 14.0 15.0

Transpose of the above matrix is:
0.0 4.0 8.0 12.0
1.0 5.0 9.0 13.0
2.0 6.0 10.0 14.0
3.0 7.0 11.0 15.0

pd@pd-IdeaPad-Gaming-3:~/G2/GPGPU-with-GLES/build$ 
```

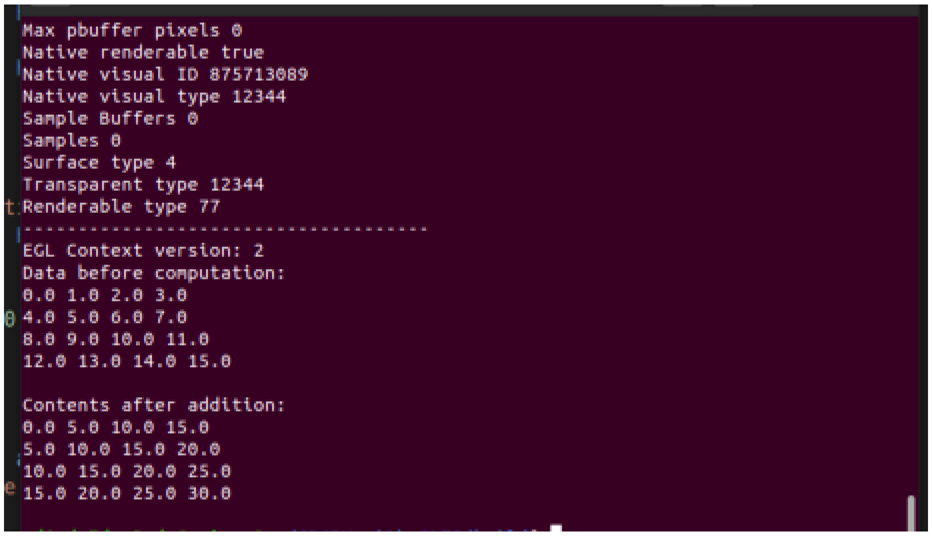
1.6. Addition of a Matrix with its Transpose

This is an extension of the above operation in which one of the matrices has its input coordinates reversed and the returning values are added with the elements of the original matrix. As usual, the final result is stored as a vec4 in gl_FragColor.

The values returned by gl_FragColor are processed by the glReadPixels function in the CPU. They are finally converted into regular floats and displayed on the screen.

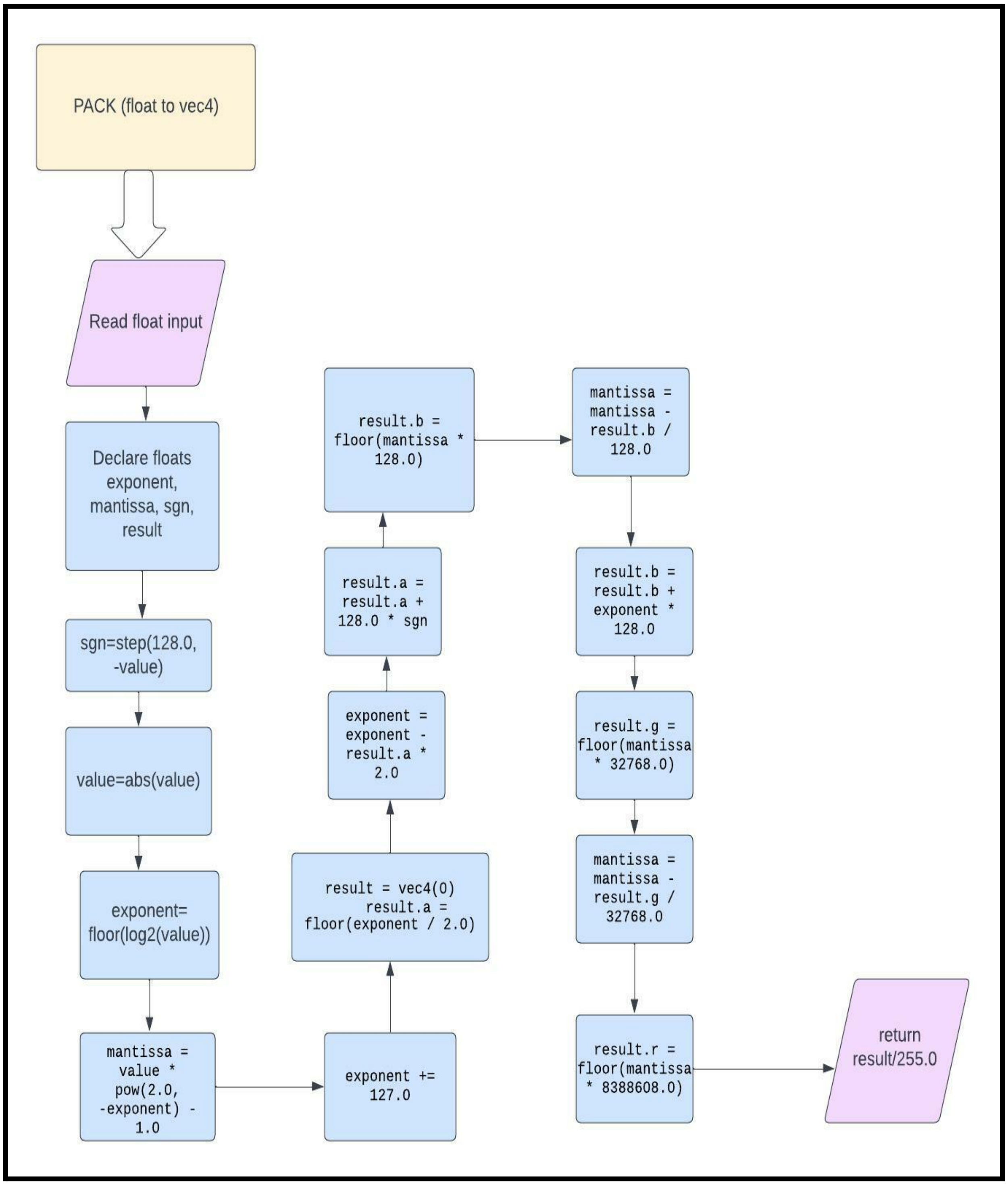
Shader Code:
https://github.com/Pratham-Bot/GPGPU-with-GLES/blob/main/shaders/mat_transpose_addition.fs

Output:



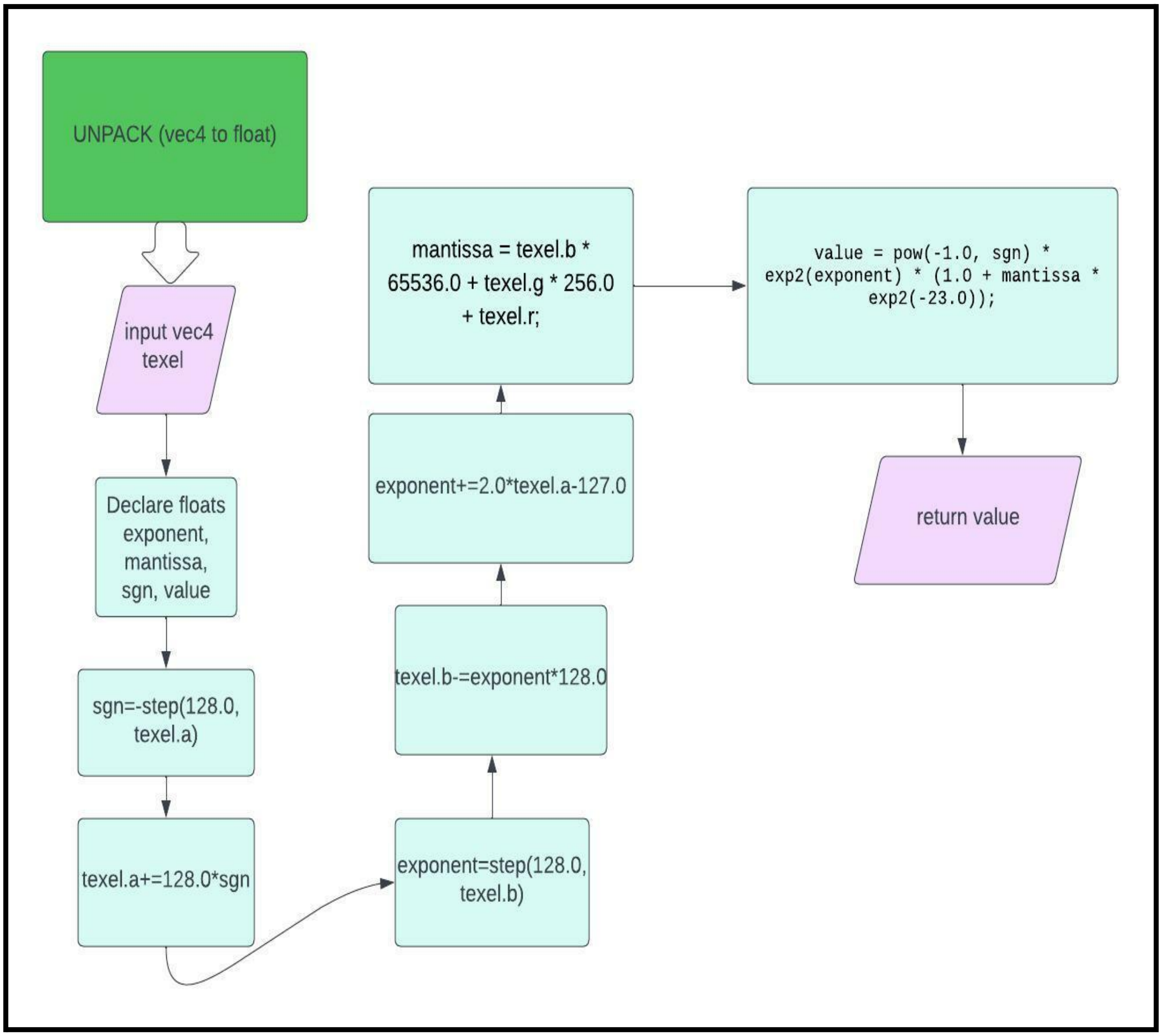
```
Max pbuffer pixels 0
Native renderable true
Native visual ID 875713089
Native visual type 12344
Sample Buffers 0
Samples 0
Surface type 4
Transparent type 12344
Renderable type 77
-----------------------------------
EGL Context version: 2
Data before computation:
0.0 1.0 2.0 3.0
4.0 5.0 6.0 7.0
8.0 9.0 10.0 11.0
12.0 13.0 14.0 15.0

Contents after addition:
0.0 5.0 10.0 15.0
5.0 10.0 15.0 20.0
10.0 15.0 20.0 25.0
15.0 20.0 25.0 30.0
```

1.7 Float to vec4 conversion



PACK (float to vec4)
Read float input
Declare floats exponent, mantissa, sgn, result
sgn=step(128.0, -value)
value=abs(value)
exponent= floor(log2(value))
mantissa = value * pow(2.0, -exponent) - 1.0
exponent += 127.0
result = vec4(0) result.a = floor(exponent / 2.0)
exponent = exponent - result.a * 2.0
result.a = result.a + 128.0 * sgn
result.b = floor(mantissa * 128.0)
mantissa = mantissa - result.b / 128.0
result.b = result.b + exponent * 128.0
result.g = floor(mantissa * 32768.0)
mantissa = mantissa - result.g / 32768.0
result.r = floor(mantissa * 8388608.0)
return result/255.0

1.8 vec4 to float conversion


UNPACK (vec4 to float)
input vec4 texel
Declare floats exponent, mantissa, sgn, value
sgn=-step(128.0, texel.a)
texel.a+=128.0*sgn
exponent=step(128.0, texel.b)
texel.b-=exponent*128.0
exponent+=2.0*texel.a-127.0
mantissa = texel.b * 65536.0 + texel.g * 256.0 + texel.r;
value = pow(-1.0, sgn) * exp2(exponent) * (1.0 + mantissa * exp2(-23.0));
return value

Section 3: Future Scope of the Project

GPUs are currently used in computing devices for rendering visual output on the screen. Sometimes, programs use a GPU to do things other than calculating graphics such as doing normal computations which can be done in parallel. These days, the hash values of electronic currencies like BitCoin are done with the help of the GPU, since such a task is easily parallelized and each calculation is rather simple.

What makes the GPUs useful for such tasks is that they typically have a larger number of processing elements (PEs) and more total GPU memory. They also have a tighter integration with the host system, in some cases, including pairing with the conventional processors but always possessing greater perks such as better support of GPU access to host memory.

The reason why GPU programming is not as prevalent might be due to the following reasons:

1. Total cost of ownership
2. Common perceptions

The less popularity of GPU programming is not due to the lack of creativity but rather about discovering the problems to solve. Often, only those problems can be solved by GPU programming where the overwhelming strength of numbers translates into improved performance.

The project can be benefitted by the following additions:

- AI compatibility on BeagleBone Black
- Using Vulkan on BeagleBone Black

General Purpose computing with the GPU can and has, to some extent, revolutionized scientific research by improving the speed and efficiency of data mining, thus, helping in the early and accurate discovery of diseases such as cancer. Machine Learning is much faster on the GPU than the CPU as they have more specialized ML hardware built into them. Thus, its usage in the Artificial Intelligence industry would be immense and GPUs are driving the future of AI-ML.
With the increase in the number of neurons in the neural networks, efficient deployment of GPUs would increase scalability, dramatically reduce the training and ROI times and lower the overall deployment of the hardware. For training neural networks for live predictions, deployment of GPUs in the cloud is the solution.
As mentioned, the GPUs are currently used to calculate the hash values of electronic currencies. The GPUs might also be used for blockchain search and to insert queries.

The future of General purpose computing with the GPU looks promising, with tech giants such as NVIDIA laying the foundations for future open and parallel code. Once harnessed properly, the GPU is a very powerful tool in every programmer’s arsenal.

References

- GITHUB REPOSITORY -
  https://github.com/JDuchniewicz/GPGPU-with-GLES

- OPEN GL Functions-
  https://docs.gl/

- Sharing Internet connections from Host to BBB using USB
  https://gist.github.com/pdp7/d2711b5ff1fbb000240bd8337b859412

- OPENGL Programming-
  https://www.linuxjournal.com/content/introduction-opengl-progra
  mming

- OPENGL installations-
  http://www.codebind.com/linux-tutorials/install-opengl-ubuntu-lin
  ux/

- Matrix Multiplication logic:
  http://www.vizitsolutions.com/portfolio/webgl/gpgpu/matrixMultip
  lication.html