PROJECT REPORT
Comparing Computation Time of Dense Matrix Operations In CPU and GPU

Prayas Patnaik
Computer Engineering
The University of Texas, Dallas

Abstract

GPU Technology is offering great prospects in computation. It was originally developed for Graphics Applications. It has been extensively used for parallel computation. The main pain point for the developers is to reduce the penalty in memory transfer. This project is based on assessing the execution time of complex matrix manipulation in CPU and GPU. We compared the obtained results to see which performs better.

1. Introduction

This project is based on addressing the limitation of GPU in real-time control. The excellent performance in the computation of GPUs is paid for a penalty in memory transfer. As a result, Applications of real-time control suffer from often unacceptable latency. To compare the performance of the CPU and GPU. We are computing the Execution time of image manipulation of different image resolutions in CPU and GPU and comparing them. OpenCV library has been used to do the manipulation in CPU and OpenGL Framework for GPU. Three operations have been performed on different image resolutions(Grey Scale Conversion, Gaussian Blur, and Image Rotation). OpenCV(Open Source Computer Vision Library) is an open-source computer vision and machine learning software library and OpenGL(Open Graphics Library) is a cross-language, cross-platform application programming interface for rendering 2D and 3D graphics. The execution time(CPU) is the sum of time taken to bring the image into the memory and the time to process it. The execution time(GPU) is the sum of the time taken to transfer the image to the GPU, the time taken to process it, and the time taken to retrieve the image back from the GPU. The GPU image manipulation has been implemented on Mesa Intel(R) HD Graphics 520 (SKL GT2). The timings are plotted for each operation for CPU and GPU.

2. Background: OpenGL

OpenGL is not a Programming language, it is a specification, developed by the Khronos Group. The specification specifies exactly what the result/output of each function should be and how it should perform. OpenGL is by itself a large state machine: a collection of variables that define how OpenGL should currently operate. The state of the OpenGL is referred to as OpenGL Context. While doing any operations we define the state of the buffers, the specifications of the image which is to be rendered on the screen and then change the state of OpenGL by changing some context variables that sets how OpenGL should draw. The first thing we need to do before sending the data to the GPU is to create an OpenGL Context and set up an application window to draw in. The state of the OpenGL is commonly referred to as OpenGL Context. We change its state by setting some options, manipulating some buffers, and then rendering using the current context. The next and the most important aspect of OpenGL is to understand the Graphics Pipeline.

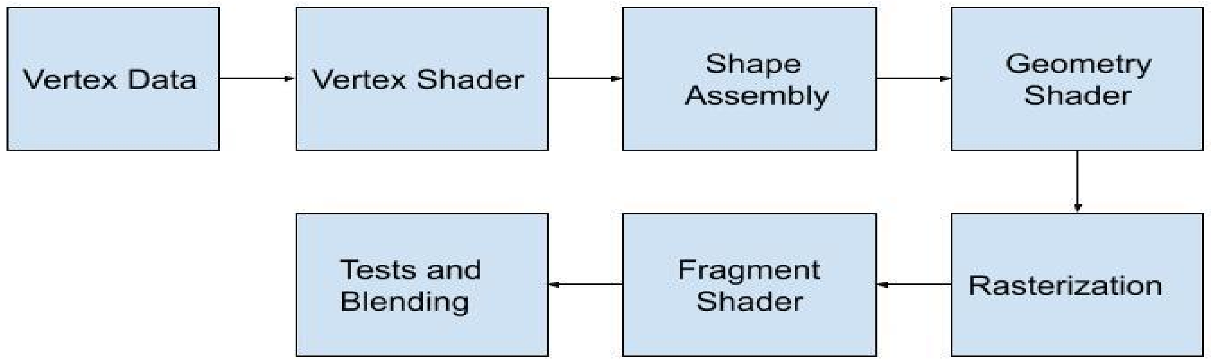
2.1 Graphics PipeLine

To render a Pixel on a screen we need to communicate with the GPU and we need a medium and that medium is OpenGL. The purpose of OpenGL is to transfer the data from the CPU to GPU. Once it reaches the GPU, it goes through the **OpenGL Graphics Pipeline.** The graphics pipeline takes 3D coordinates as inputs and transformed these 3D coordinates into colored 2D pixels on the screen. The graphics pipeline can be divided into several steps which require the output of the previous steps as its input. These steps can be executed in parallel. Because of this parallel nature, graphics cards have thousands of cores to process the data within the graphics pipeline. The processing cores run small programs on the GPU for each step of the pipeline. Shaders are these small programs that actually run on the GPU. Some of the shaders are configurable by the developer. Vertex shader and Fragment shader has been written separately for each of the different operations.

Let's talk about the Data transfer from CPU to GPU. GPU requires three sets of Data Attributes, Uniforms, and Textures.

Attributes are used by the GPU to assemble a geometry and apply lightning. The common attributes are vertex positions, Normal Coordinates, and U-V Coordinates. Vertex positions are used by the GPU to create the geometry of the figure. Normals are vectors perpendicular to a surface and are used to apply **Light** to the image. The U-V coordinates are used to map an image to a different image.

Uniforms are another way to pass data from our application on the CPU to the shaders in the GPU. It is slightly different from the vertex attributes. It is global. It means that a uniform variable is unique per shader program object and can be accessed from any shader at any stage in the shader program. A **Texture** is a 2D Image. It is used to store a large collection of data to send to the shaders.



OpenGL Graphics Pipeline

The first part of the pipeline is the **Vertex Shader** which takes vertices as input and transforms them into different 3D coordinates. It allows us to do different basic processing on the vertex attributes. Vertex Attribute is an input variable to a shader that is supplied with per-vertex data.

The **Shape Assembly stage** takes input from all the vertices from the Vertex Shader that form a primitive and assemble all the points in the primitive shape.

The output of the Shape Assembly stage is passed to the **Geometry shader**. The Geometry Shader takes input from a collection of vertices that form a primitive and has the ability to generate other shapes by joining new vertices to form new primitives.

The output of the geometry shader is then passed to the **Rasterization Stage** where it maps the resulting primitive(s) to the corresponding pixels on the Final screen, resulting in fragments for the fragment shader to use. Clipping is performed before Fragment shaders.

The main function of the **Fragment Shaders** is to calculate the final color of a pixel. It has the data about the scene and uses it to calculate the final color of the pixel.

The last stage is Alpha Test and Blending stage. The stage checks the corresponding depth value of the fragment and uses it to check if the fragment is in front or behind other objects and should be discarded accordingly.

3. Implementation

We have used five different Image resolutions for comparisons(1024*768,1280*1024,280*720,1920*1080) OpenCV Library has been used to do the computation in the CPU and OpenGL Specifications have been used to do Computations in the GPU. A python script is used to plot the values generated from the code. Implementation detail will be described in further sections

3.1 OpenCV Functionality

It has been implemented as a Cpp file. There are three switch-case statements that decide the operations which are to be performed on the Images. The default case function shows the image as it is.

Three Functions have been defined:

- GreyConversionOperationFunction
- GaussianBlurOperationFunction
- RotateImageOperationFunction

Let's talk about the functions separately -

- **GreyConversionFunction** is a function which reads the image and transforms it into a greyscale image using OpenCV in-built function cv::CvtColor(Mat inputImag , Mat outputImage, cv::COLOR_BGR2GRAY). To measure the mean execution time to transform the image into greyscale, 50 iterations have been taken to calculate the meantime.

  Execution Time = Time taken to bring the image into memory + Color Converting the image to Gray

  We can find the time taken by different parts of a program by using the std::chrono Library in C++. std::chrono has two distinct objects-timepoint and duration. A TimePoint represents a point in time whereas a duration represents an interval or span of time.The library allows us to subtract two-time points to get the interval of time passed in Between. The std::chrono provides us with three clocks with varying accuracy. The High_resolution_clock is the most accurate and hence it has been used to measure the Execution time.

- **GaussianBlurOperationFunction** is a function that reads the image and performs Gaussian Blur on the image using a 5*5 kernel using OpenCV in-built function GaussianBlur(Mat inputImage , Mat outputImage , Size(5,5),0). 50 iterations have been used to calculate the meantime.

  Execution Time = Time taken to bring the image into memory + Gaussian Blur on the Image

  In the Gaussian Blur operation, the image is convolved with a Gaussian Filter. The Gaussian filter is a low-pass filter that removes the high-frequency component in the Image.

- **RotateImageOperationFunction** is a function that reads the image and rotates the image by the angle specified as an input to the function. It involves two steps. The first is to calculate the rotation matrix and the second is to warp the image with the rotation matrix.
  The rotation matrix can be calculated by the Opencv in-built function.

  - getRotationMatrix2D(Point Centre, double rotationAngle , double scale)
  - Centre Point can be calculated by dividing the image size by 2.
  - WarpAffine(Mat inputImage , Mat rotationMatrix , Size image)

  WarpAffine does Affine Transformation. It is a transformation that can be expressed in
  The form of a matrix Multiplication.

  Execution Time = Time taken to bring the image into memory + Affine Transformation on the image

3.2 OpenGL Functionality

It has been implemented as a CPP file. It has three switch cases-statements That decide the shader which is to be implemented on the GPU. Different shaders have been written for each of the Image Operations. The default shader does not do any operation. A shader is a program that actually runs on a GPU.

Shaders are as follows:

- shaderGaussian.shader
- shaderGrey.shader
- shaderRotation.shader

Let’s talk about the shader separately -

- **Gaussian Blur Shader**

  - shaderGaussian.shader applies Gaussian blur on the image. It contains two shaders Vertex Shaders and Fragment Shaders. Vertex Shader takes the position and texture coordinates and passes the texture coordinates to the Fragment Shaders.

  - Fragment Shader does the Gaussian Blur on the image. A Gaussian Blur is based on the Gaussian Curve. The Gaussian Curve has a larger area close to its center, using its values as weights to blur an image gives more natural results. In the Fragment Shader, we have implemented a two-pass Gaussian blur. We first do a horizontal blur with the wights on the scene texture and then on the resulting texture do a vertical blur. Shaders run on every pixel of the image. We have used the texture function to do the sampling of the texels in both horizontal as well as vertical directions.

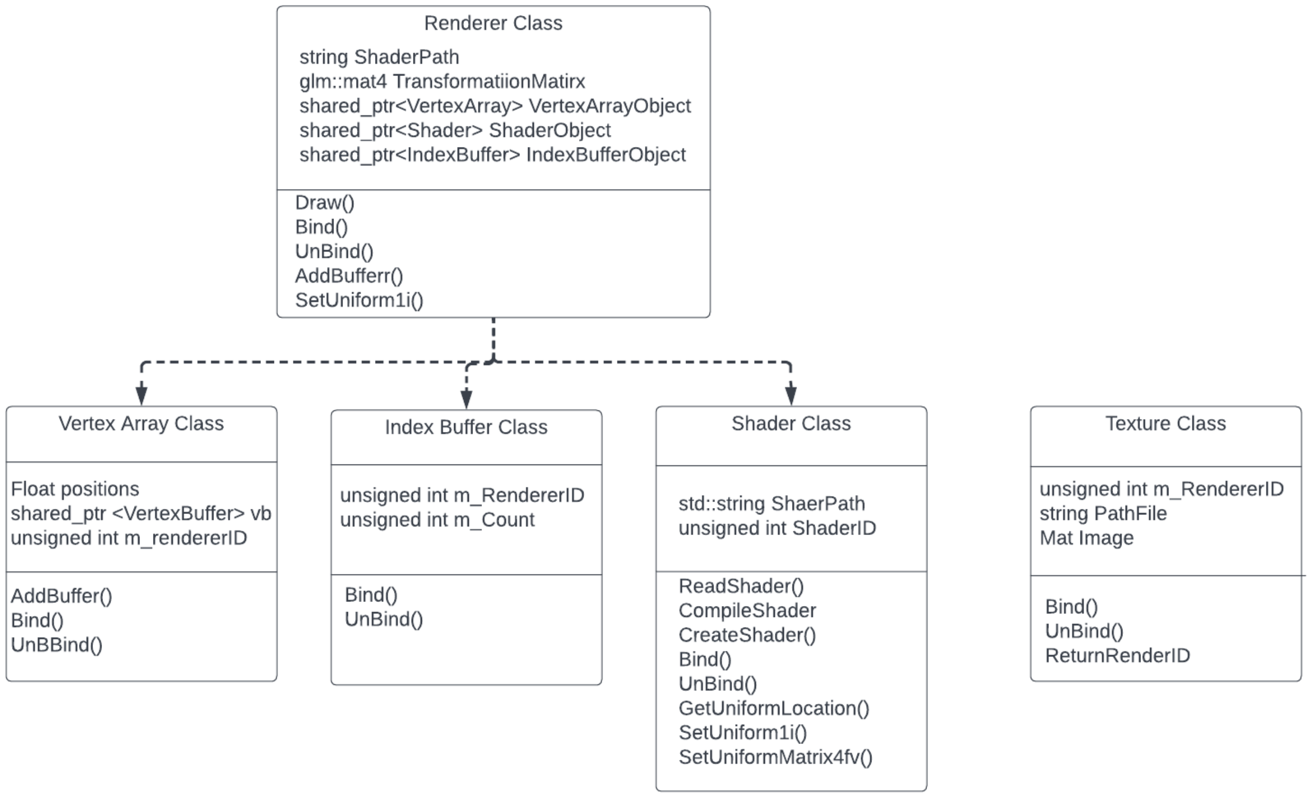
- **Grey Scale Shader**

    - ShaderGrey.shader applies grayscale conversion to the image. The vertex shader is the same as that of the vertex shader of the Gaussian Shader.
    - In the fragment shader, we sample the texture and average the intensities of the r channel, g channel, and b channel and set the mean value to the resultant image channels.

- **Rotation Shader**

    - The rotation shader rotates the image by the angle specified by the user. In the vertex Shader, we have defined a uniform mat4 transform which is a transformation or a rotation matrix. This matrix is multiplied by the vertex coordinates of the image.
    - The fragment shader only samples the texels of the texture.

3.3 Class Diagram



Renderer Class

It instantiates VertexArray, shader, and IndexBuffer class. The constructor of the Renderer class creates VertexArrayOject, ShaderObject, and IndexBufferObject.

It has five-member functions:

- Draw - Draw function calls the glDrawElements function which renders primitives from array data. It specifies multiple geometric primitives with very few subroutine calls. We pre-specify the states of vertex buffers, textures, and colors. We use them to construct a sequence of primitives with a single call to glDrawElements.

- Bind - Bind function internally calls the bind function of the VertexArrayObect, ShaderObect, and IndexBufferObject. Whenever we start an operation that uses OpenGL's state, we bind the object with the defined setting. In this, we have set the Vertex Array, ShaderObject, and Index Buffer settings.

- UnBind - It is the opposite of bind. It detaches the object from the current context.

- AddBuffer - It calls the member function Add a buffer of Vertex Array Class. It creates a vertex Buffer Object that stores the vertices in the GPU's memory. The advantage of the buffer object is that we can send large batches of data all at once to the graphics card. As Sending data to the graphics card from the CPU is relatively low. We always try to send as much data as possible at once. There is a unique ID associated with VertexBuffer. We use this ID to bind the Vertex Buffer object and then copy the user-defined data into the currently bounded buffer and we can tell OpenGL how it should interpret the vertex Data using glVertexAttribPointer. This function takes the position of the different attributes of the vertex and stores it. When we need to use these different attributes, we can enable this by using the glEnableVertexttribArray function. This AddBuffer function encapsulates the above functionality.

- SetUniformli - This function set the values of the texture Coordinates or Transformation matrix defined or any other variable mentioned in the shader.

Shader Class

Shader class basically read the shader code, compile it and attach it to the program object. Let's talk about the implementation of the member function of the Shader Class.

It has 9 member functions:

- ReadShader - The main objective of this function is to read the shader file in which the vertex shader and the Fragment shader have been defined and store the contents in a string.

- CompileShader - We have written the source code for the shaders but in order for OpenGL to use the shader it has to dynamically compile it at run-time from it source code. We create a shader object and this object is referenced by an ID. we attach the shader source code to the shader object and compile the shader. To check if the compilation has been successful or not, we checked for compile-time errors as well in the function.

- CreateShader - This function basically creates a Program object and calls the Compile Shader function. The CompileShader function is called for both Vertex Shader and Fragment Shader. It returns an ID for both the shaders. We attach the shader to the program object and then link it. After successfully attaching and linking the shader code to the program object, we delete the shader objects. It is not required anymore.

- Bind - It tells the OpenGL to use the defined program object as a part of the current rendering state.It uses glUseProgram(int programObjectID) function to do that.

- UnBind - It unbinds the program object which was earlier bound to OpenGL context.
- GetUniformLocation - It returns the location of the uniform defined in the shader.
- SetUniform1i - It sets the value of the uniform in the shader.
- SetUniformMatrix4fv - It also sets the value of the uniform in the shader, the data type of the uniform is different. In this function, we set the 4*4 matrix uniform.

IndexBuffer Class

This class is implemented to reduce the memory overhead. Let's understand the concept of Indexing. Suppose we want to draw a rectangle. We can draw a rectangle using two triangles. We define three vertices to draw a triangle. We have to use 6 vertices to draw a rectangle(2 Triangles). But two vertices would be the same in the two triangles and we are unnecessarily using the memory to store those values. We can define a rectangle using 4 points only. To do that, we use an index buffer object. They are like vertex buffer object, that stores the indices that OpenGL uses to decide what vertices to draw. This is called indexing drawing.

It has 2 member functions:

- The constructor of this class Creates the Generates the index buffer and binds it and put the index array into the memory of the GPU.
- Bind - It binds the allocated Index buffer to the current rendering state.
- UnBind - It unbinds the Index Buffer and detaches it.

Vertex Array Class

It creates a Vertex Array Object and can be bound like a vertex Buffer Object and any subsequent vertex attribute calls from that point on will be stored inside the Vertex Array Object. We create VAO using OpenGL functions glGenVertexArrays and bind it with the current OpenGL context, then we create Vertex Buffer objects in the AddBuffer function and put data in the vertex buffer object. Now it's easier to switch between different Vertex Data and attributes configurations. When we want to draw, we have to bind the vertex Array object and draw the call function.

It has 3 member functions:

- Add Buffer - This function creates the Vertex Buffer object and put data into it.
- Bind - It binds the Vertex Array Object to the current rendering state.
- UnBind - It unbinds the Vertex Array Object from the current rendering state.

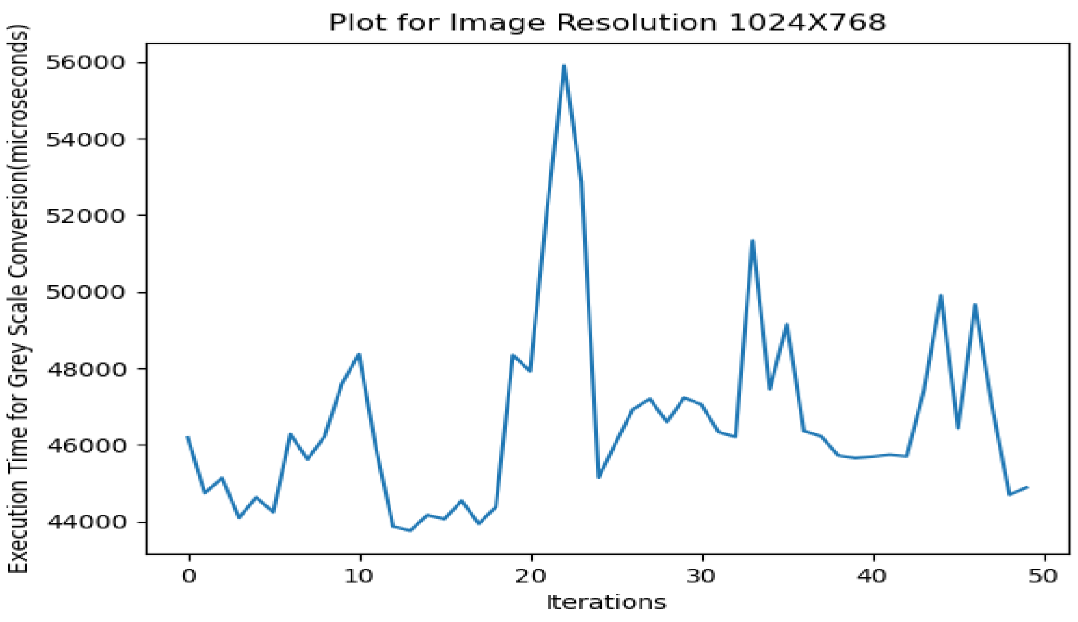
Texture Class

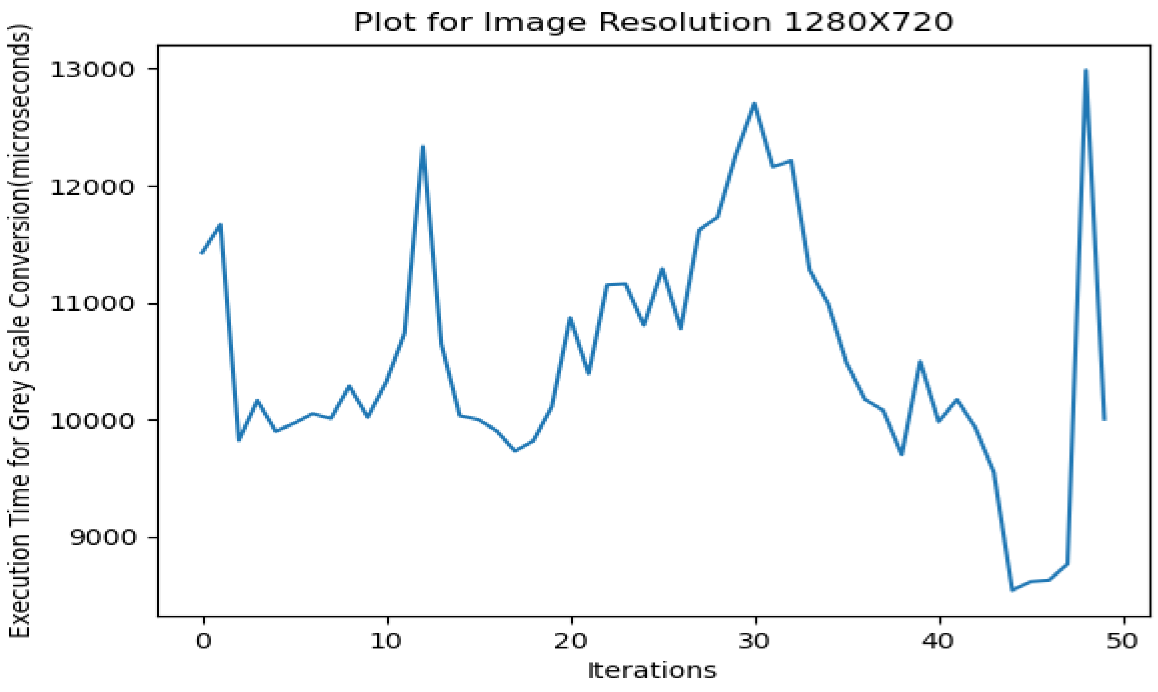
A texture is a 2D image that is used to add details to an object. In order to map a texture to an object, we need to tell which part of it corresponds. Each Vertex has a texture coordinate associated with them that specifies what part of the texture image to sample from. Retrieving the texture color using texture coordinates is called sampling.

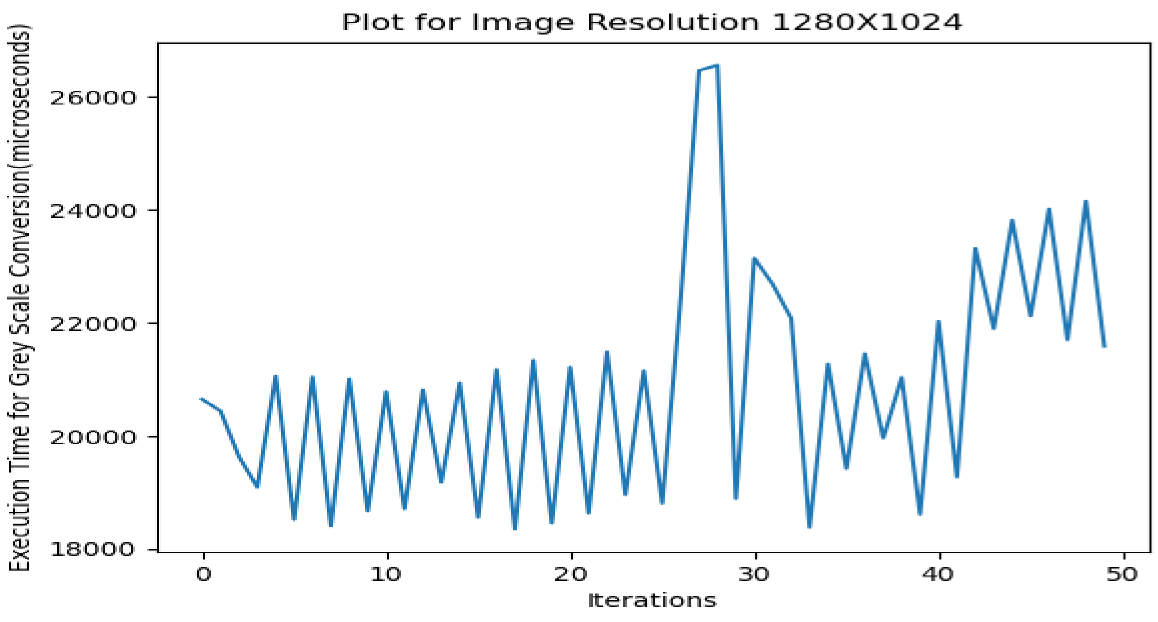
It has 2 member functions:

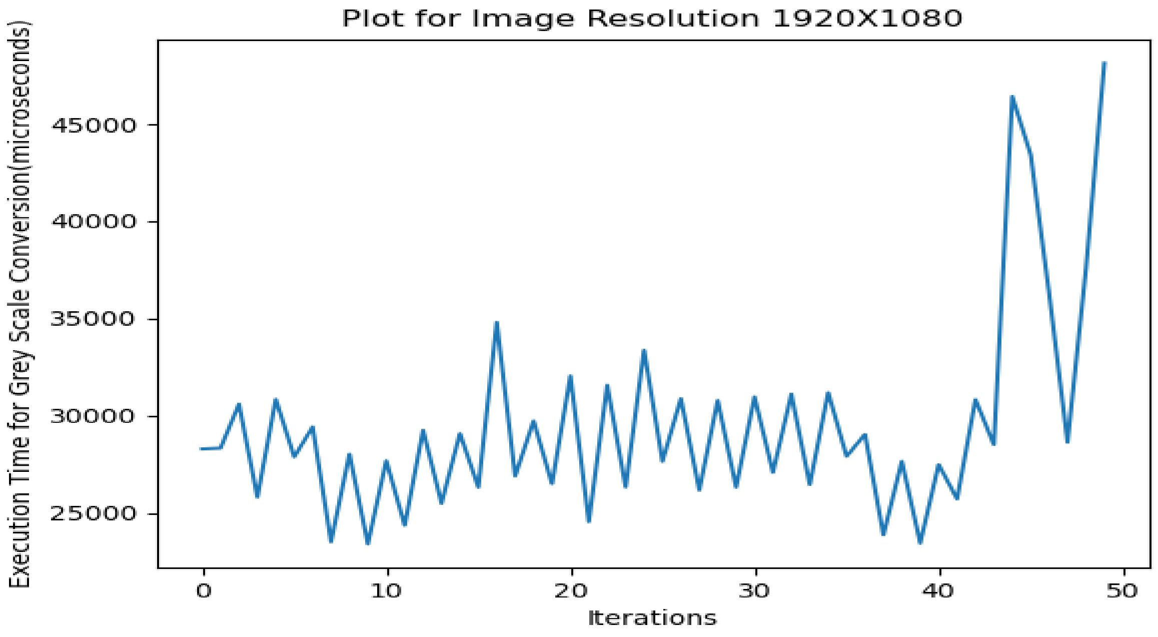
- Bind - It binds the texture with the current rendering state.
- UnBind - It unbinds the texture with the current rendering state.

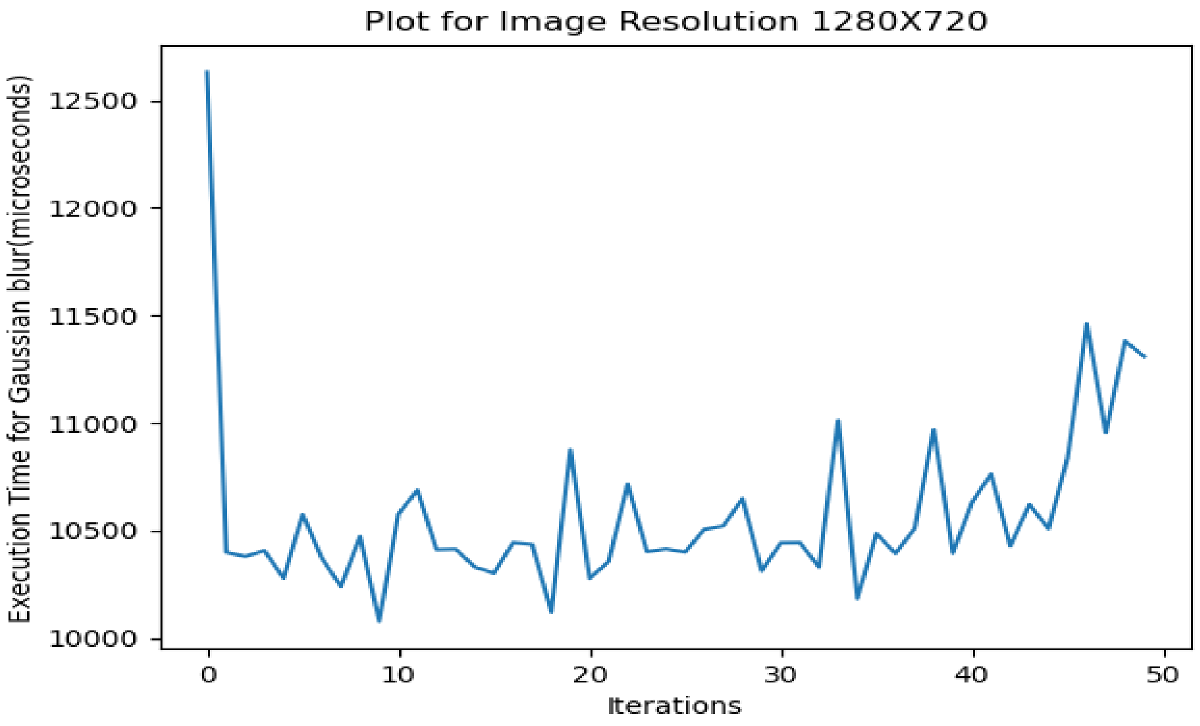
4. OpenCV Plots

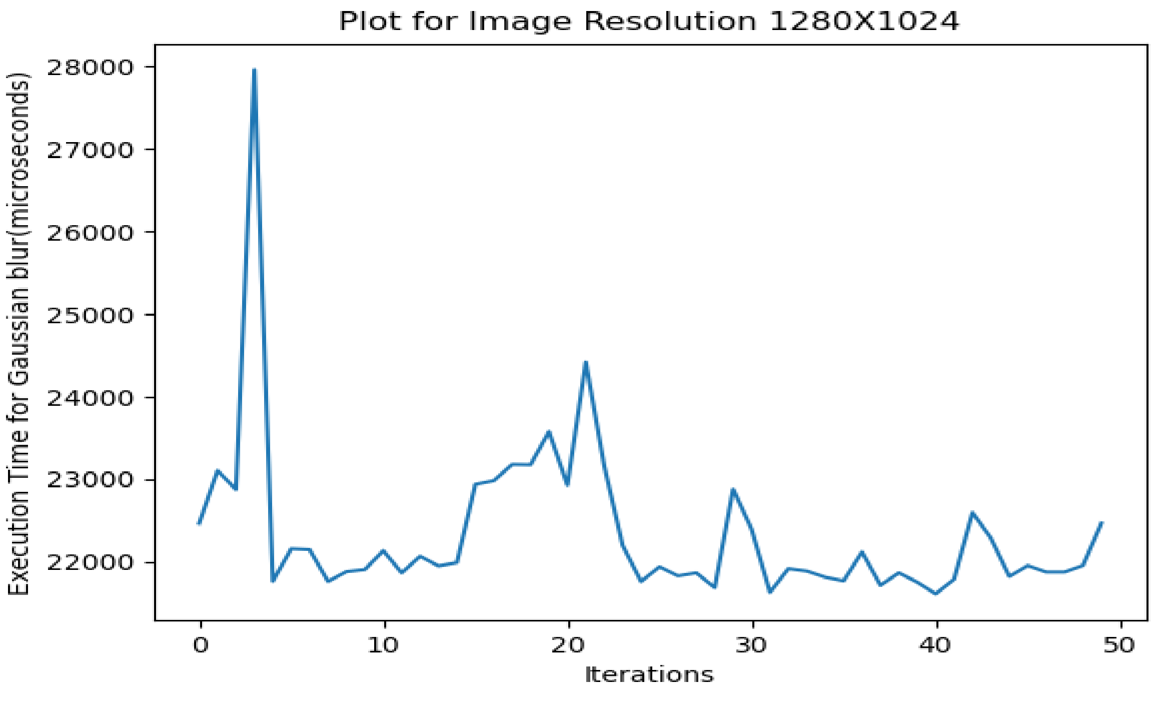


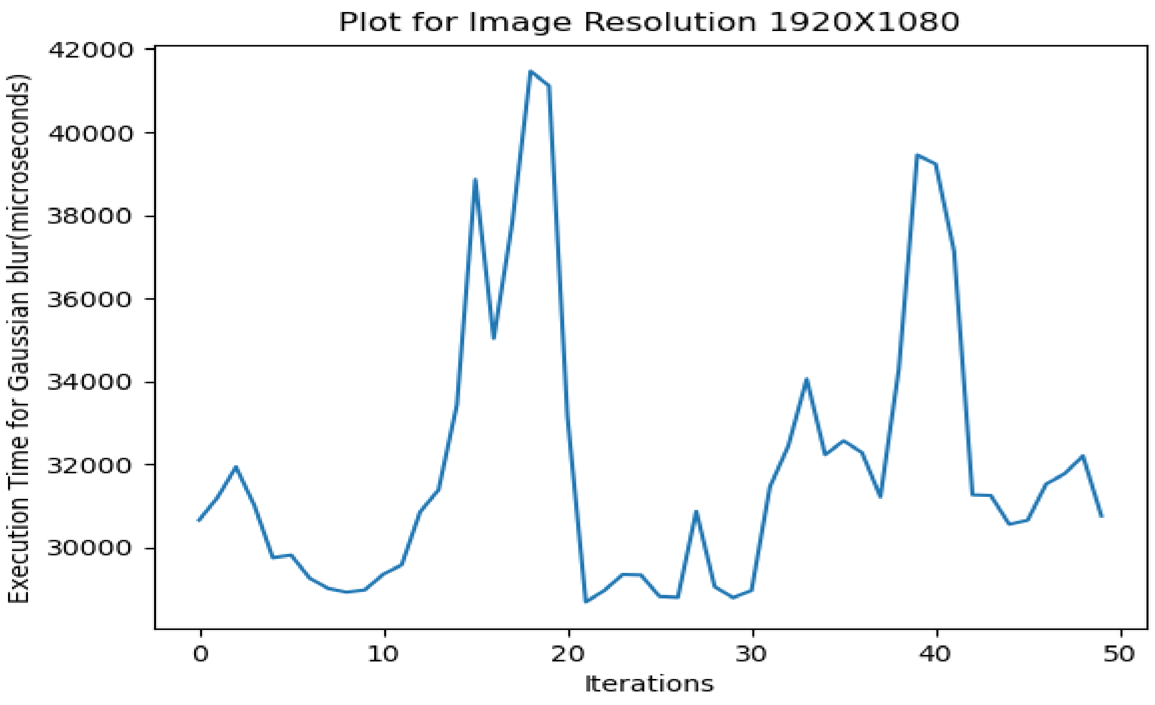


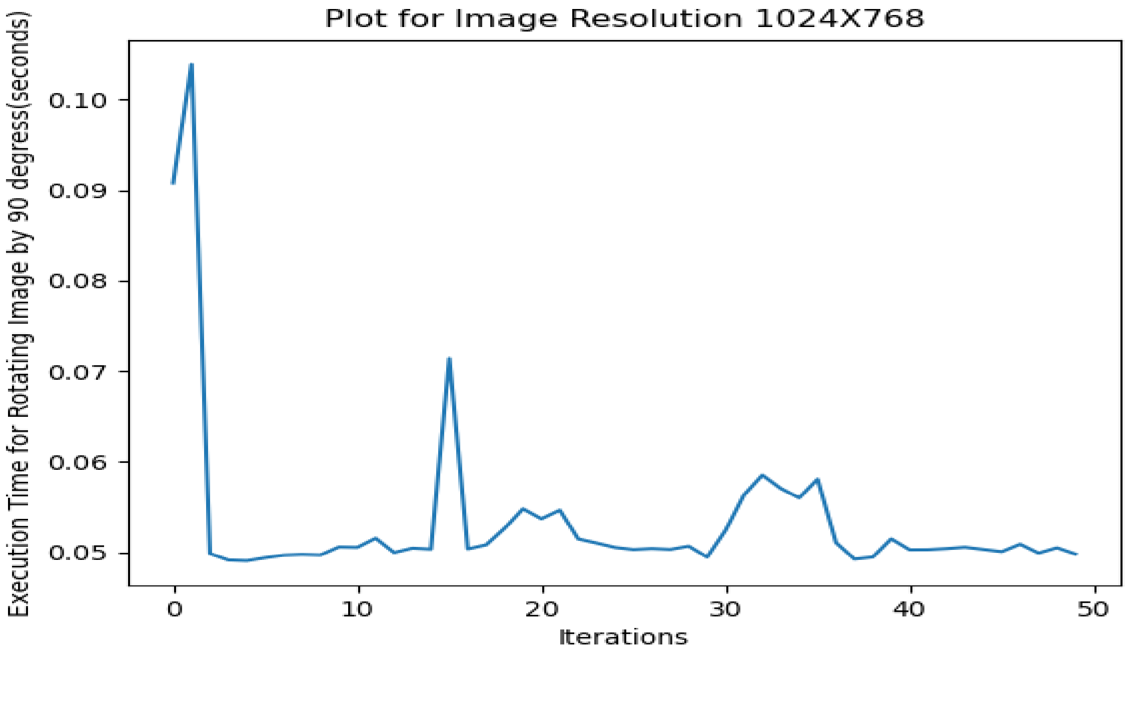

Plot for Image Resolution 1280X1024
Execution Time for Grey Scale Conversion(microseconds)
Iterations
18000
20000
22000
24000
26000
0
10
20
30
40
50

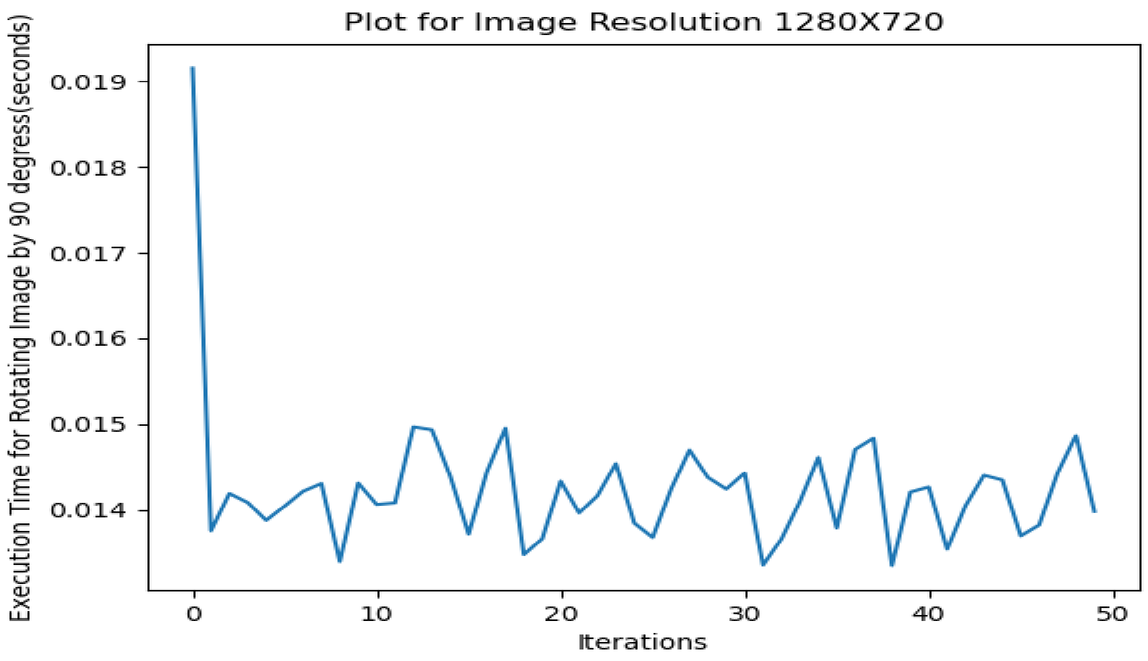

Plot for Image Resolution 1920X1080
Execution Time for Grey Scale Conversion(microseconds)
Iterations
25000
30000
35000
40000
45000
0
10
20
30
40
50

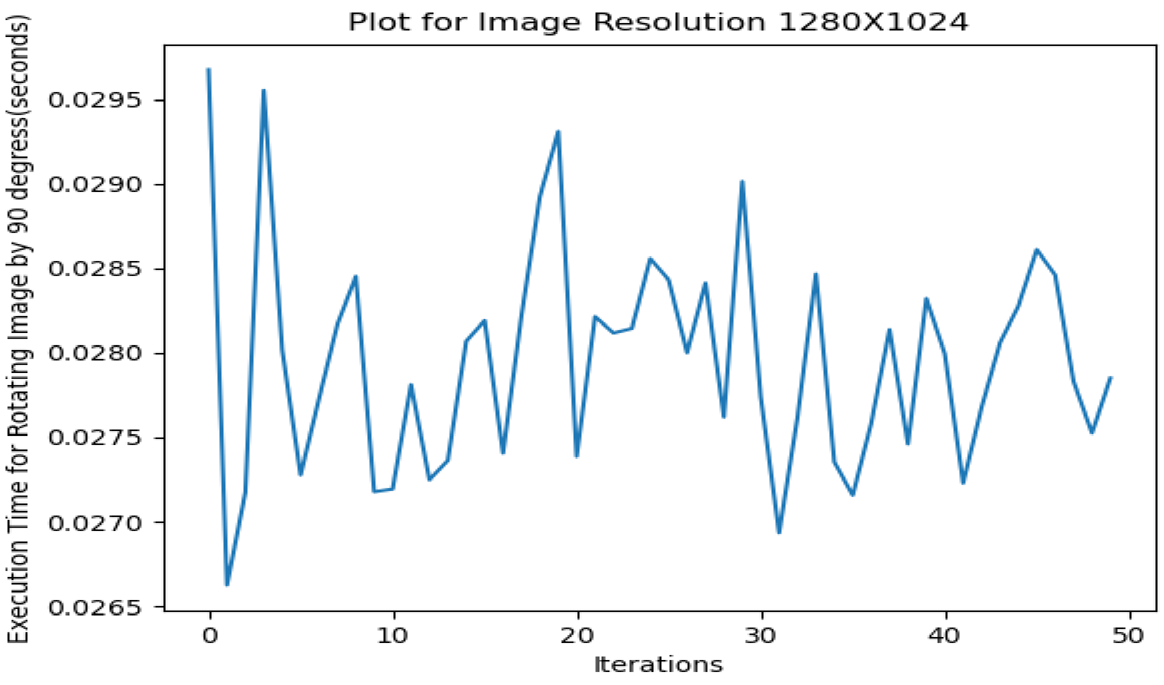

Plot for Image Resolution 1280X720
Execution Time for Gaussian blur(microseconds)
Iterations
12500
12000
11500
11000
10500
10000
0
10
20
30
40
50

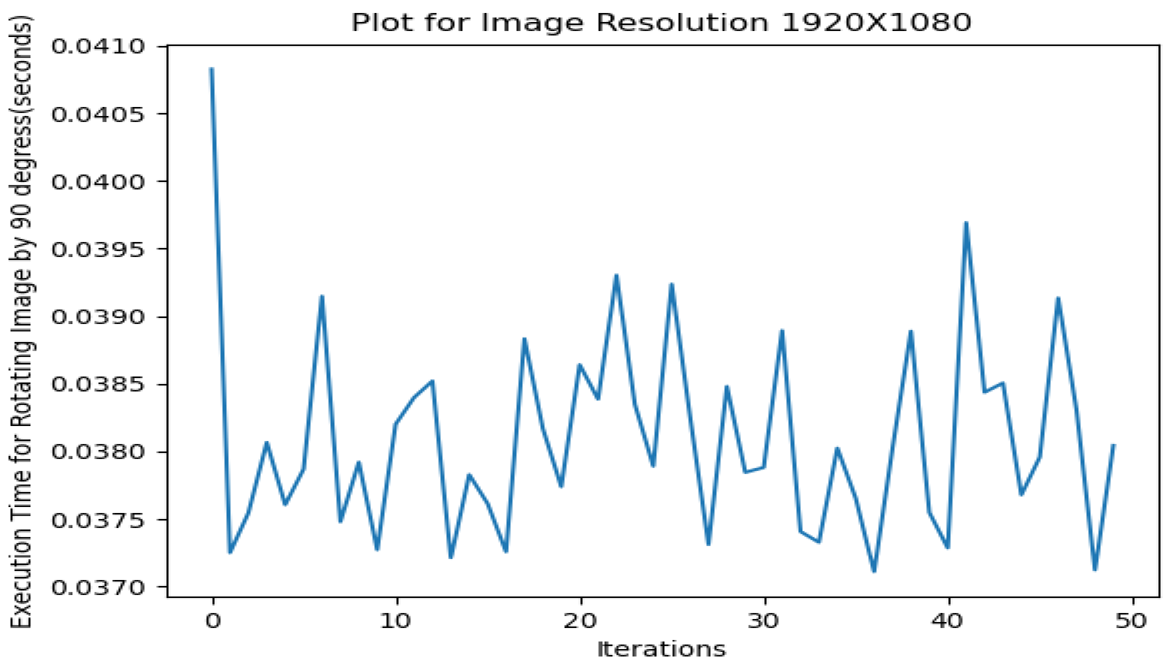

Plot for Image Resolution 1280X1024
Execution Time for Gaussian blur(microseconds)
Iterations
28000
27000
26000
25000
24000
23000
22000
0
10
20
30
40
50


Plot for Image Resolution 1920X1080
Execution Time for Gaussian blur(microseconds)
42000
40000
38000
36000
34000
32000
30000
0
10
20
30
40
50
Iterations

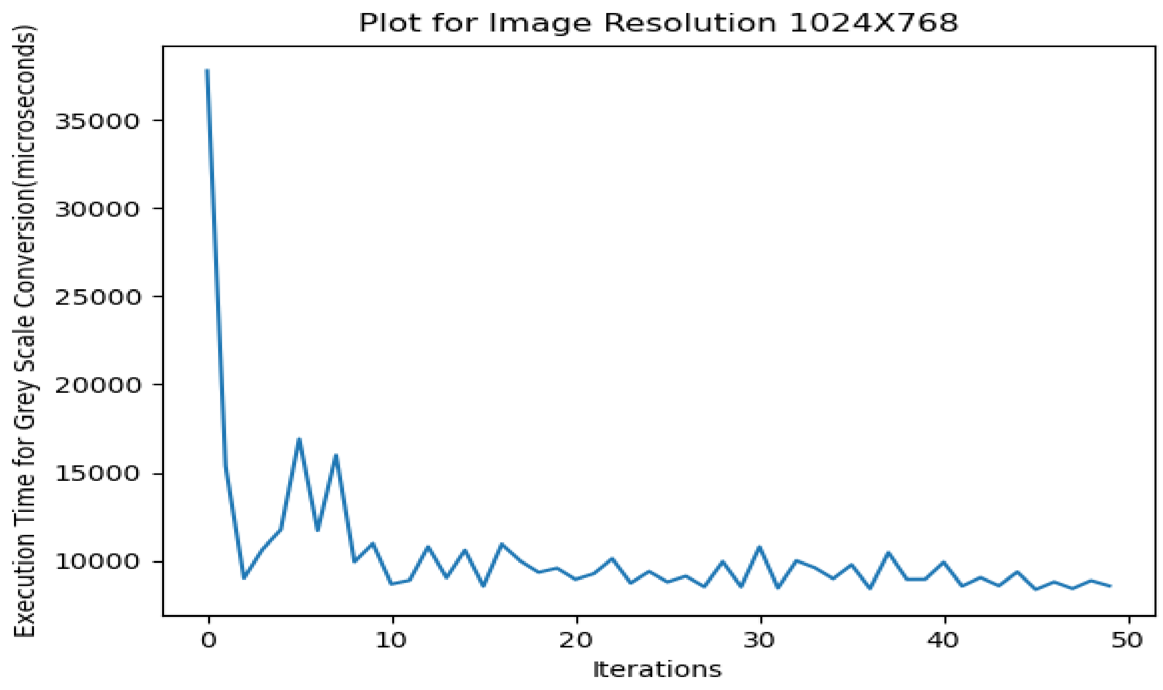

Plot for Image Resolution 1024X768
Execution Time for Rotating Image by 90 degress(seconds)
0.10
0.09
0.08
0.07
0.06
0.05
0
10
20
30
40
50
Iterations

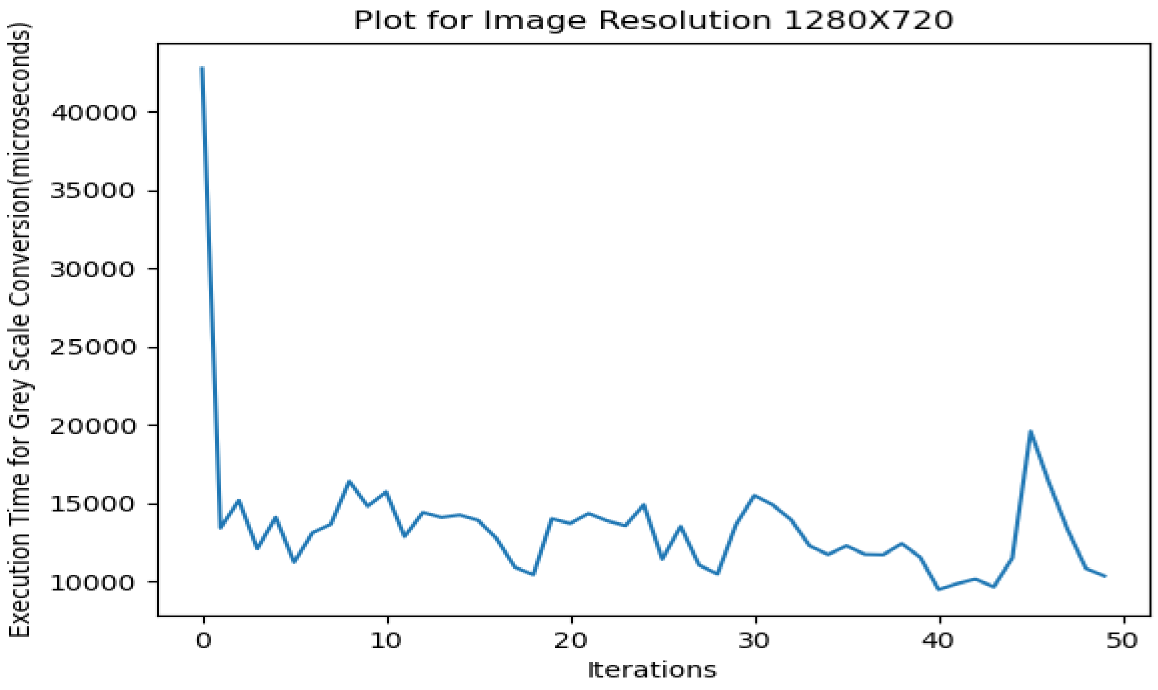

Plot for Image Resolution 1280X720
Execution Time for Rotating Image by 90 degress(seconds)
0.019
0.018
0.017
0.016
0.015
0.014
0
10
20
30
40
50
Iterations

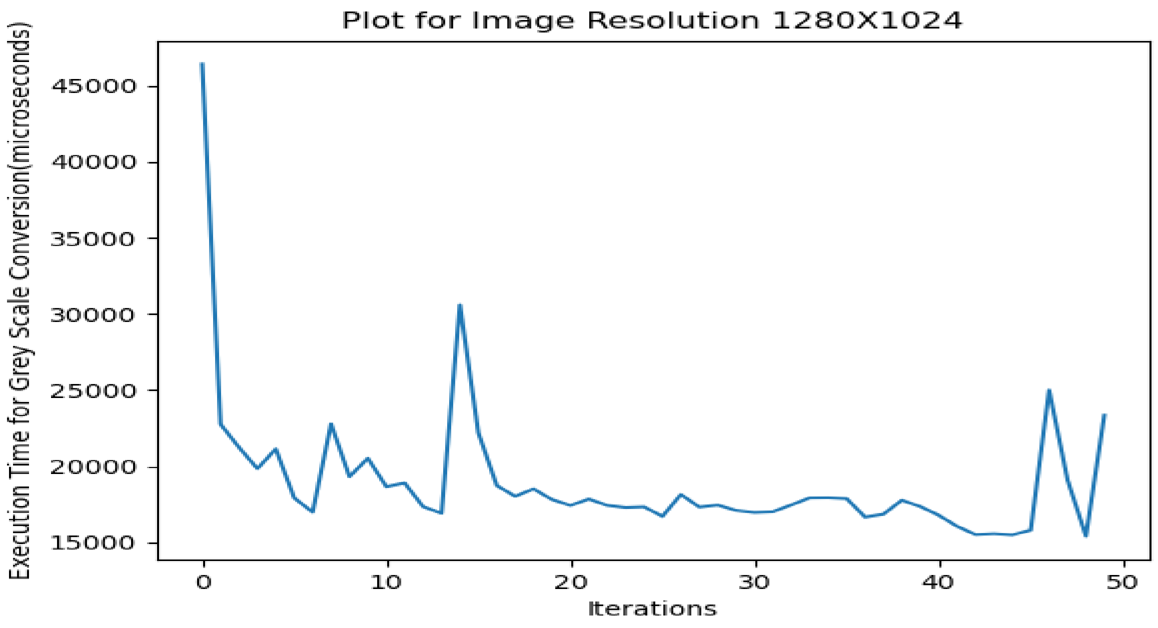

Plot for Image Resolution 1280X1024
Execution Time for Rotating Image by 90 degress(seconds)
0.0295
0.0290
0.0285
0.0280
0.0275
0.0270
0.0265
0
10
20
30
40
50
Iterations

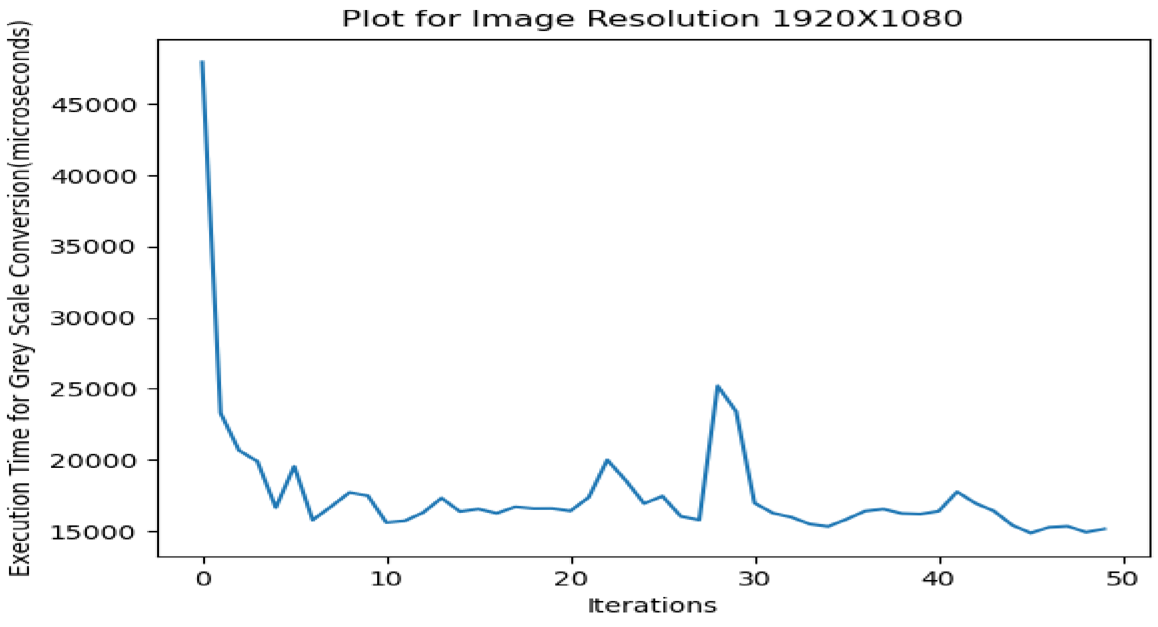


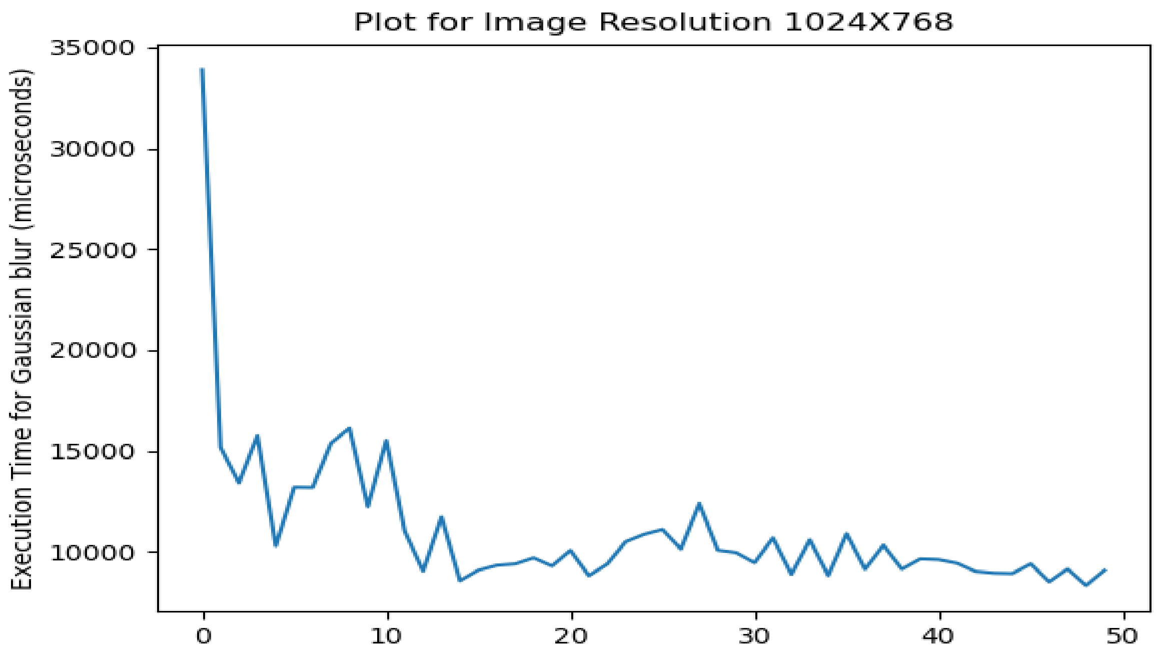
5. OpenGL Plots

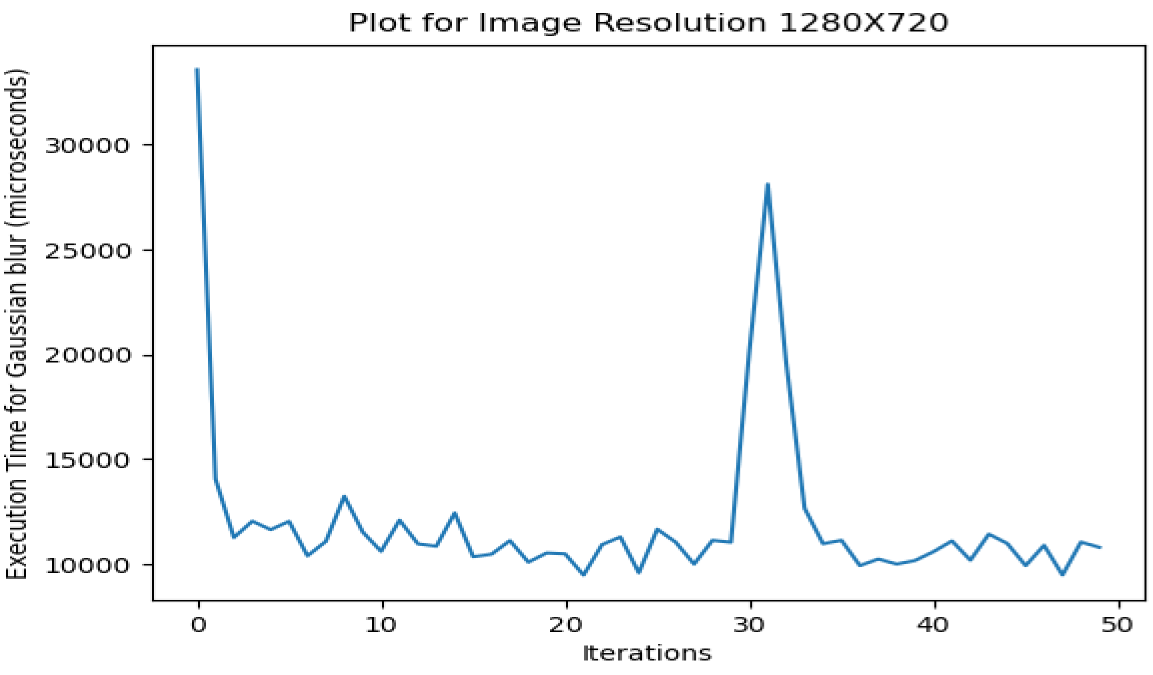


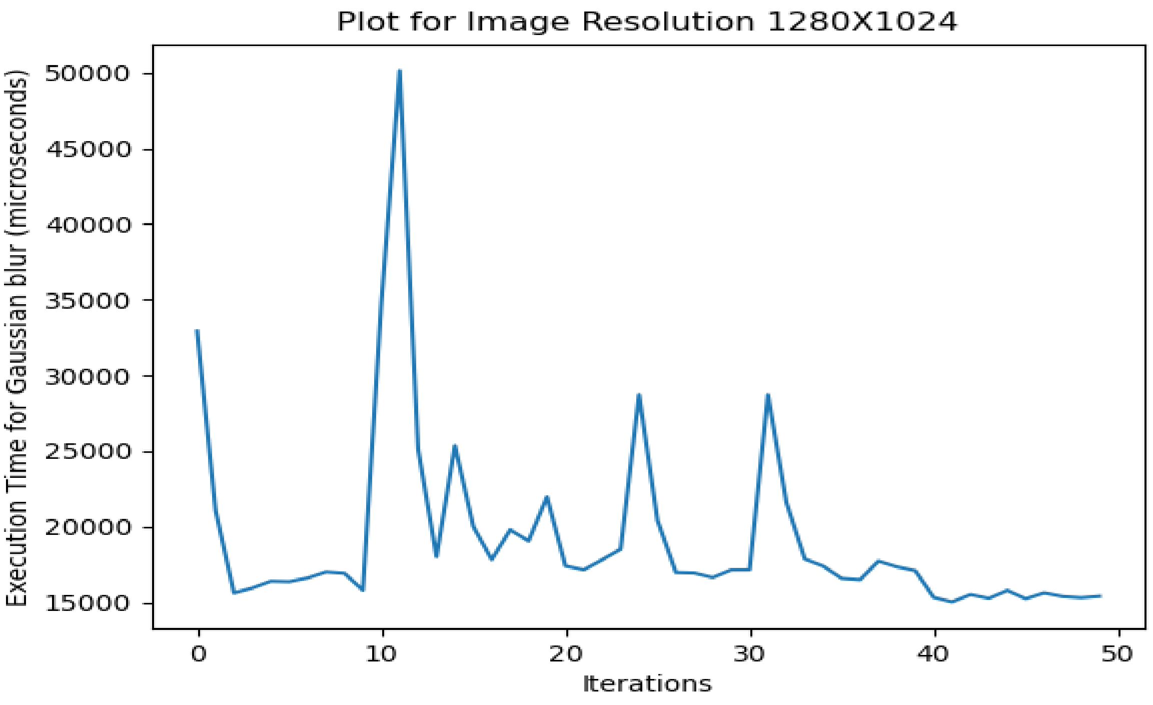

Plot for Image Resolution 1280X720
Execution Time for Grey Scale Conversion(microseconds)
40000
35000
30000
25000
20000
15000
10000
0
10
20
30
40
50
Iterations

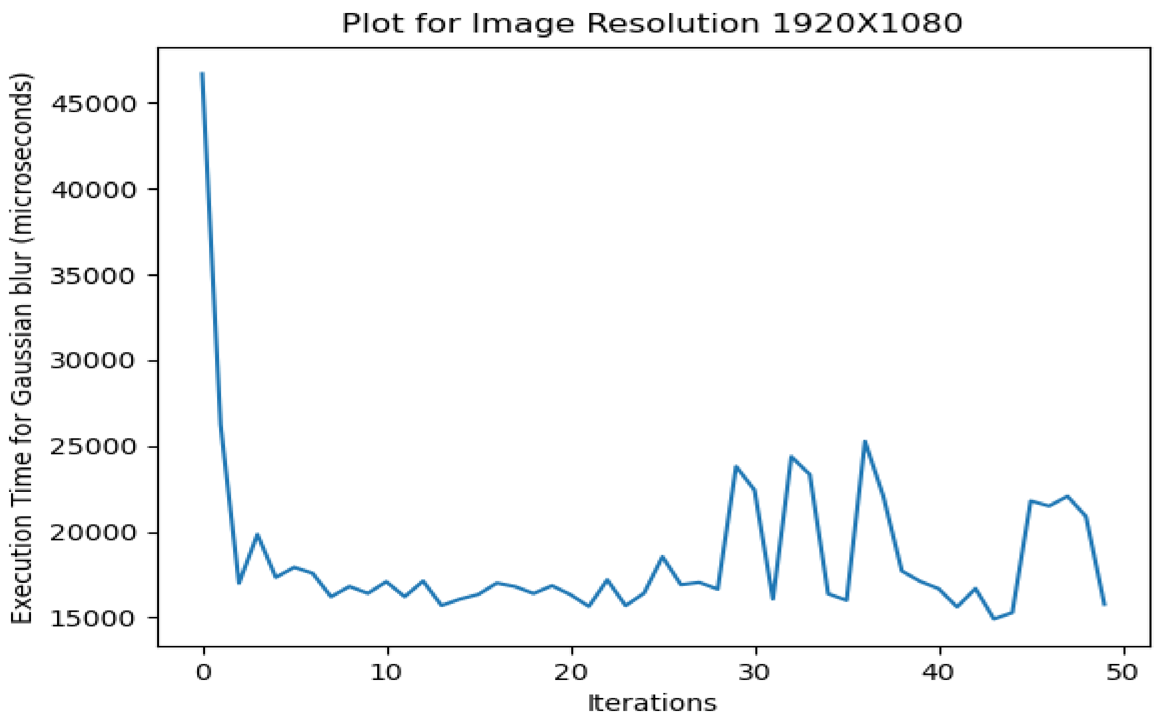

Plot for Image Resolution 1280X1024
Execution Time for Grey Scale Conversion(microseconds)
45000
40000
35000
30000
25000
20000
15000
0
10
20
30
40
50
Iterations

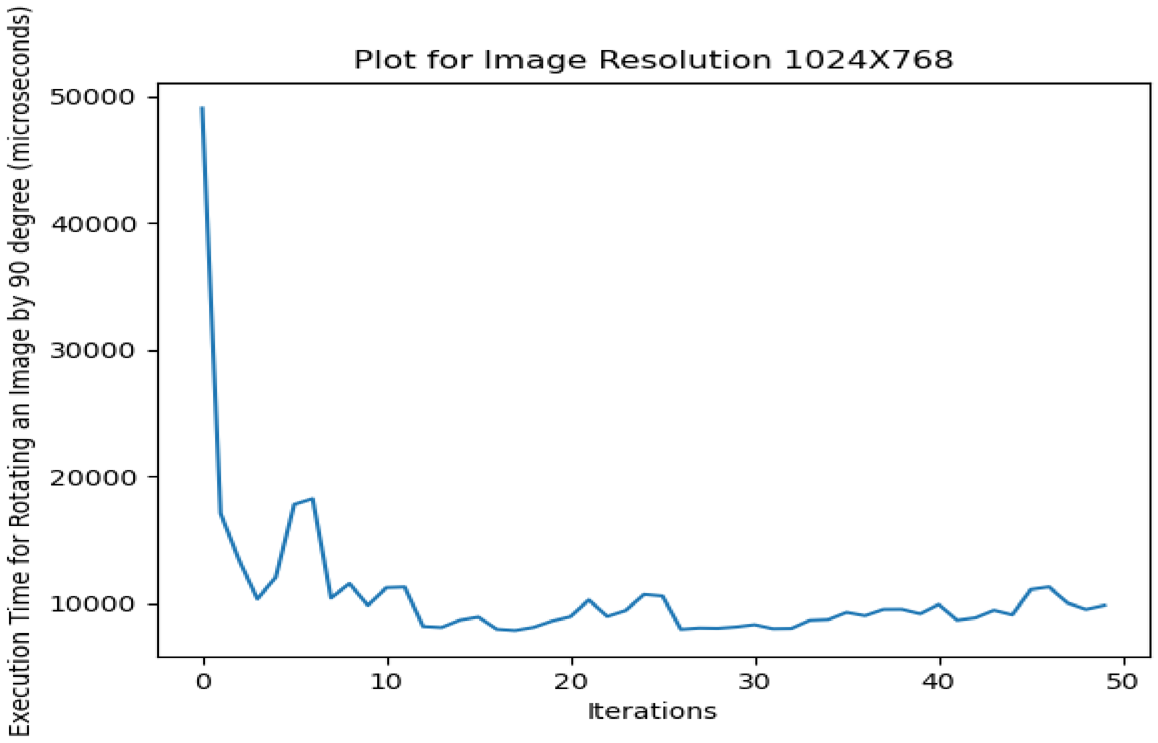

Plot for Image Resolution 1920X1080
Execution Time for Grey Scale Conversion(microseconds)
45000
40000
35000
30000
25000
20000
15000
0
10
20
30
40
50
Iterations

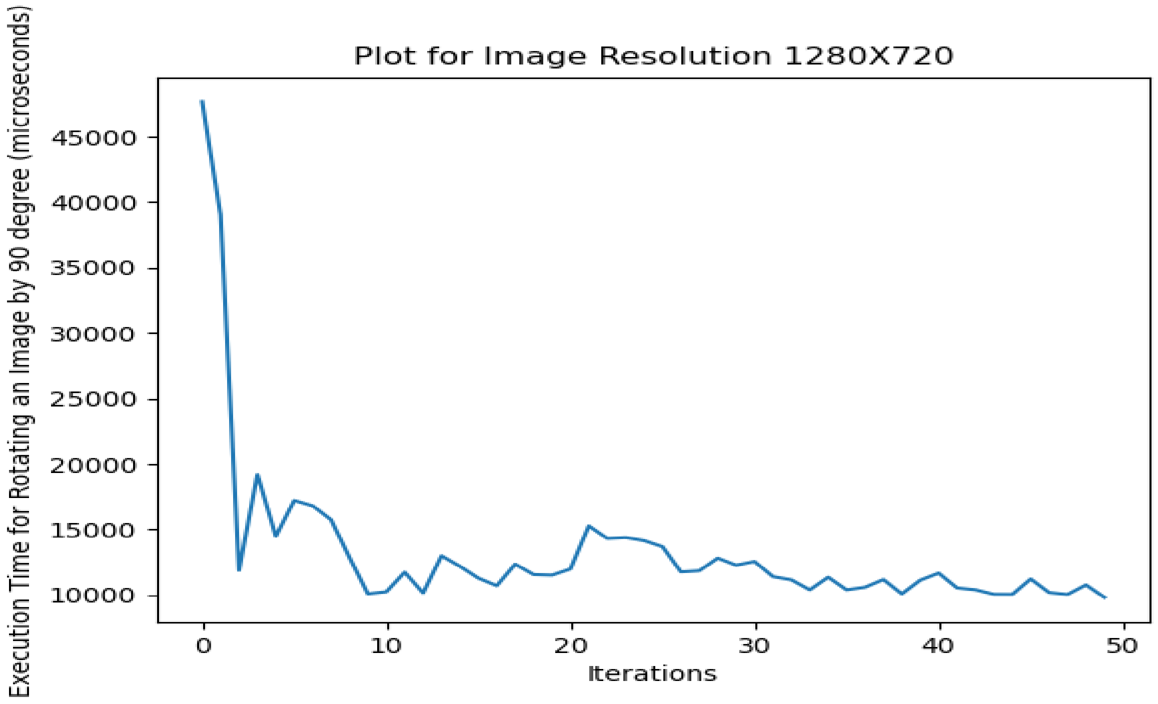

Plot for Image Resolution 1024X768
Execution Time for Gaussian blur (microseconds)
35000
30000
25000
20000
15000
10000
0
10
20
30
40
50
Iterations


Plot for Image Resolution 1280X720
Execution Time for Gaussian blur (microseconds)
30000
25000
20000
15000
10000
0
10
20
30
40
50
Iterations

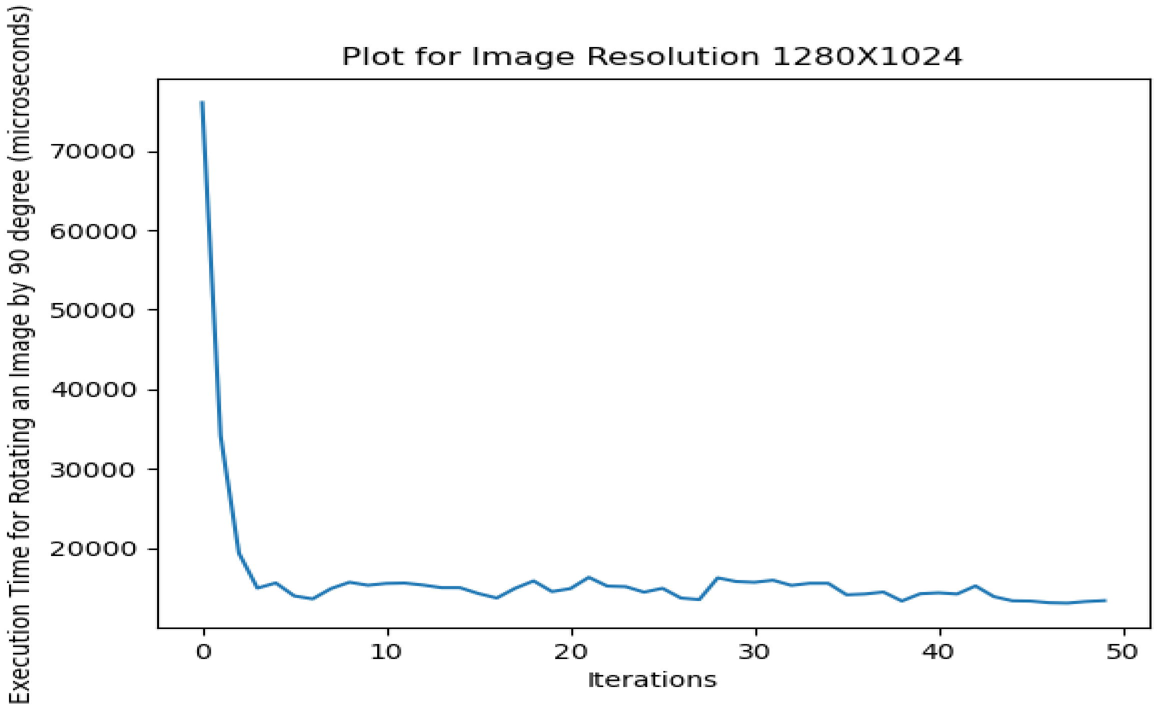

Plot for Image Resolution 1280X1024
Execution Time for Gaussian blur (microseconds)
50000
45000
40000
35000
30000
25000
20000
15000
0
10
20
30
40
50
Iterations


Plot for Image Resolution 1920X1080
Execution Time for Gaussian blur (microseconds)
45000
40000
35000
30000
25000
20000
15000
0
10
20
30
40
50
Iterations

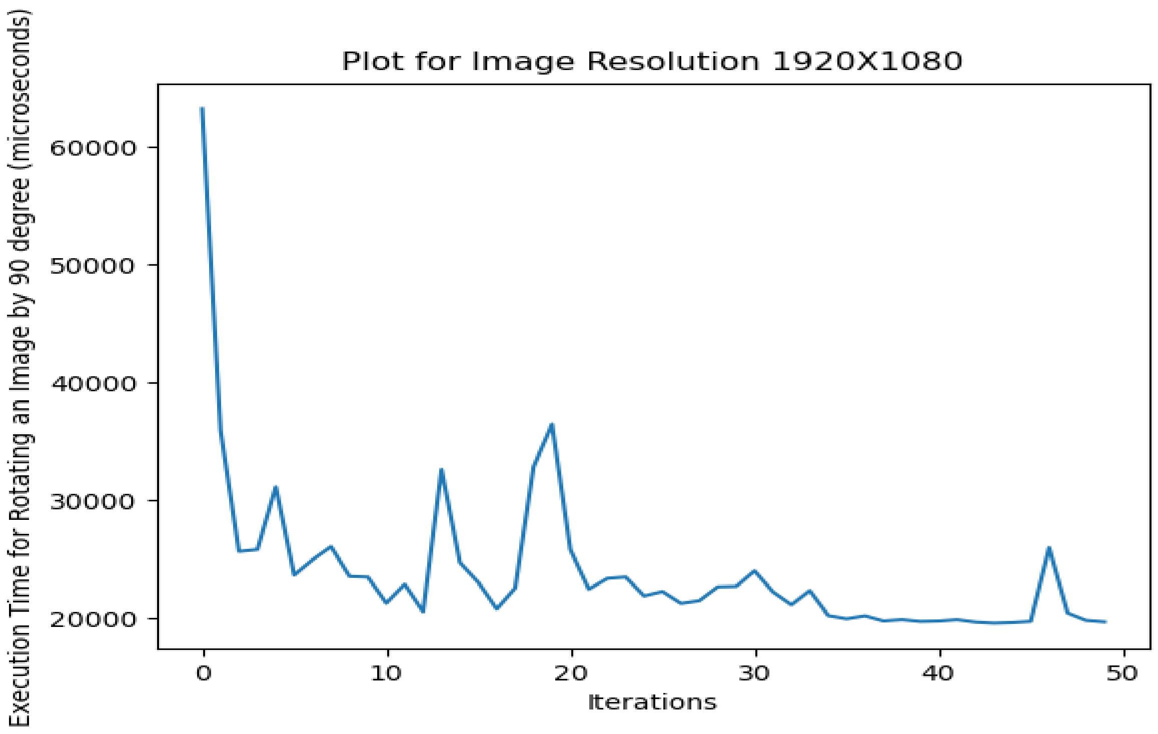

Plot for Image Resolution 1024X768
Execution Time for Rotating an Image by 90 degree (microseconds)
50000
40000
30000
20000
10000
0
10
20
30
40
50
Iterations



Plot for Image Resolution 1280X720
Execution Time for Rotating an Image by 90 degree (microseconds)
Iterations
45000
40000
35000
30000
25000
20000
15000
10000
0
10
20
30
40
50



Plot for Image Resolution 1280X1024
Execution Time for Rotating an Image by 90 degree (microseconds)
Iterations
70000
60000
50000
40000
30000
20000
0
10
20
30
40
50



6. OpenCV Results

(Table showing Mean, Median ,Standard Deviation of Execution Time of Grey scale Operation)

Image Resolution	**Mean(us)**	**Median(us)**	**Standard Deviation(us)**
1024*768	**46655.7**	**46217**	**2465.28**
1280*1024	**20868.12**	**21022.5**	**2010.5**
1280*720	**10531.34**	**10232**	**1025.66**
1920*1080	**29538.6**	**28297**	**5238.99**

(Table showing Mean, Median ,Standard Deviation of Execution Time of Gaussian blur Operation)

Image Resolution	**Mean(us)**	**Median(us)**	**Standard Deviation(us)**
1024*768	**48051.14**	**46627**	**4453.77**
1280*1024	**22354.26**	**21956.5**	**1003.506**
1280*720	**10580.36**	**10442.5**	**419.10**
1920*1080	**32011.02**	**31210.0**	**3396**

(Table showing Mean, Median ,Standard Deviation of Execution Time of Rotation(90 degrees) Operation)

Image Resolution	**Mean(us)**	**Median(us)**	**Standard Deviation(us)**
1024*768	**53572.52**	**50486**	**9834.2**
1280*1024	**27955.08**	**27997**	**656.32**
1280*720	**14247.68**	**14195.5**	**826.77**
1920*1080	**38107.28**	**37990.5**	**747.50**

7. OpenGL Results

(Table showing Mean, Median ,Standard Deviation of Execution Time of Grey Scale Operation)

Image Resolution	**Mean(us)**	**Median(us)**	**Standard Deviation(us)**
1024*768	**10411.02**	**9297.5**	**4351.66**
1280*1024	**19049.46**	**17787**	**4813.76**
1280*720	**13583.58**	**13362.5**	**4671.62**
1920*1080	**17731.34**	**16480**	**4866.54**

(Table showing Mean, Median ,Standard Deviation of Execution Time of Gaussian blur Operation)

Image Resolution	**Mean(us)**	**Median(us)**	**Standard Deviation(us)**
1024*768	**11059.3**	**9837**	**3898.8**
1280*1024	**19330**	**17152.5**	**6288.75**
1280*720	**12130**	**11011.5**	**4375.142**
1920*1080	**18668.08**	**16946**	**4972.23**

(Table showing Mean, Median ,Standard Deviation of Execution Time of Rotation(90 degrees) Operation)

Image Resolution	**Mean(us)**	**Median(us)**	**Standard Deviation(us)**
1024*768	**10720.62**	**9372.0**	**6004.30**
1280*1024	**16470.2**	**15022.5**	**9082.46**
1280*720	**13341.78**	**11610.5**	**6576.79**
1920*1080	**24073.12**	**22422.5**	**6978.03**

8. Artifact

The full implementation is available at https://github.com/prayaspatnaik21/GPU_CPU_RealTimeSystem_Performance

9. Observations

In all of the operations OpenGL has performed better than OpenCV. The execution time of OpenGL is less than that of OpenCV. The standard deviation of the execution time is quite high for both the OpenCV and OpenGL operations.

10. Conclusion

While CPUs are more widely being used for general computing, GPUs are becoming popular for organizations looking for high-performance computing. It has High Data Throughput. It consists of hundreds of cores performing the same operations on multiple data items in parallel. It can push vast volumes of processed data through a workload, speeding up specific tasks beyond what a CPU can handle. It is excelling in extensive calculations with numerous similar operations, such as matrix manipulation, etc.