OpenGL Based Testing Tool Architecture for Exascale Computing

Muhammad Usman Ashraf *m.usmanashraf@yahoo.com*
Faculty of Information and Computer Technology
Department of Computer Science
King Abdulaziz University
Jeddah, 21577, Saudi Arabia

Fathy Elbouraey Eassa *fathy55@yahoo.com*
Faculty of Information and Computer Technology
Department of Computer Science
King Abdulaziz University
Jeddah, 21577, Saudi Arabia

Abstract

In next decade, for exascale high computing power and speed, new high performance computing (HPC) architectures, algorithms and corrections in existing technologies are expected. In order to achieve HPC parallelism is becoming a core emphasizing point. Keeping in view the advantages of parallelism, GPU is a unit that provides the better performance to achieve HPC in exascale computing system. So far, many programming models have been introduced to program GPU like CUDA, OpenGL, and OpenCL etc. and still there are number of limitations for these models that are required a deep glance to fix them. In order to enhance the performance in GPU programming in OpenGL, we have proposed an OpenGL based testing tool architecture for exascale computing system. This testing architecture detects the errors from OpenGL code and enforce to write the code in accurate way.

Keywords: Exascale Computing, OpenGL, OpenGL Shading Language, GPU, CUDA, Parallelism, Exaflops.

1. INTRODUCTION

This guideline is used for all journals. These are the manuscript preparation guidelines used as a In computing system, Exascale brings up to a computing technology that has ability to achieve the performance in excess to one exaflop [6]. The current technology has capability of performance in petaflops range. The object to enhance the computing performance presents a number challenges at both software and hardware levels. To make possible to targeted flops, many computer companies are working on both software and hardware level to increase the computing performance by implementing on-chip parallelism.

Since last decade, Graphics Processing Unit (GPU) based parallel computing technologies have brought up an extensive popularity for high performance (HPC). These technologies including Compute Unified Device Architecture (CUDA), OpenGL, and OpenMP etc. have opened many new research directions and visions for future Exascale computing system. However, at the programming level there are still a number of major challenges that should be fixed by introducing the new algorithms and architectures [4].

In this paper, we have emphasized on OpenGL software interface to program a GPU. Basically, before GPU processing, there is a pipeline having number of stages. Some of those are programmable and some are non-programmable or configurable stage in pipeline [3]. Our focus is on programmable steps using OpenGL Shading Language that are discussed in section II.

Further, writing program in GLSL we found number of common error those occurred at programming level and on the bases of these found errors we proposed a testing tool architecture to increase the performance and accuracy in program. This proposed architecture provides guarantee that there will be no error in the code once a developer follow this specific technique.

Further, rest of the paper is organized in such a way that, section II consist of briefly description of basic architecture of a system with GPU presence. Section III describes the classification of programmable and non-programmable stages through pipeline in OpenGL. In section IV we have explained the number of errors found during writing program in GLSL with help of code. In section V, we have presented a testing tool architecture and its behavior. Section VI consist the tool architecture evaluation description and then conclusion and future directions is presented in last section VII.

2. GPU WITHIN MACHINE

This guideline is used for all journals. These are the manuscript preparation guidelines used as a In modern computers, GPU acts as a second computer. Similar to CPU It also has its own memory and processors. The CPU get input from user and classified either it's related to GPU or CPU itself for processing. In case of GPU processing, it forward the information to GPU for processing through a programming language like OpenGL, OpenCL or CUDA. GPU process the task using its own RAM and send the processed information back to CPU for further utilizing [2].

However a GPU is designed to perform multiple kinds of tasks including graphics rendering, geometric calculations, complex mathematical calculations etc. Moreover, GPU is a basic unit for parallel processing to accomplish high performance computing. A basic architecture is presented in figure1.1 showing how GPU is resided in a computer system.

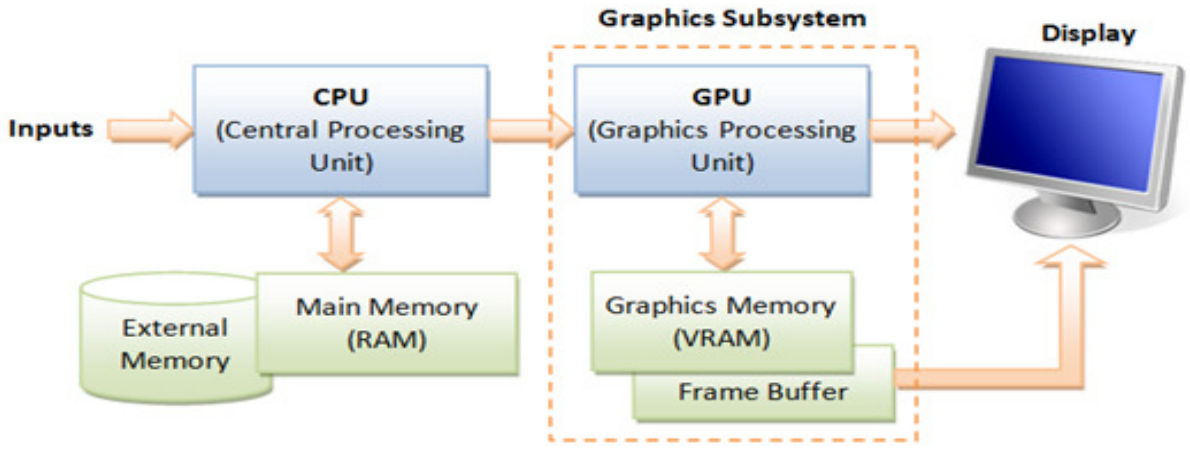

FIGURE 1: Computer system with GPU.

Figure 1 clearly showing that in a computer system CPU and GPU are almost similar but having different kind of operation processing.

3. GRAPHICS RENDERING PIPELINE

A pipeline consists of number of steps that are processed in series in such a way that one stage gets an input and provides output to next stage. Similar to this computing terminology, graphics rendering pipeline also behaves in same way as shown in figure2. This rendering pipeline gets the raw vertices and primitives as an input in 3D form. Vertex processing is the first step that gets this raw data, processes it and sends transformed vertices & primitive's data as input to rasterizer. Rasterizer further process data and convert each primitive into set of fragments. Further, fragment processor process each fragment by adding colors, positions, normal values

etc. and generate as input data for output merging step. Continuing, the pipeline steps, output merging aggregates the fragments of all primitives from 3D to 2D shapes for the display. Finally, the output object is usable for GPU as input data. In modern GPU, the pipeline stages are categorized into two types as follows:

- Programmable
- Non-Programmable.

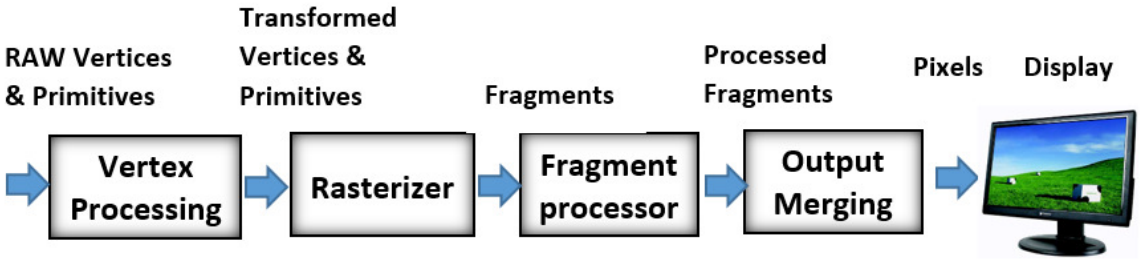

FIGURE 2: Graphics Rendering Pipeline [5].

Vertex processing and fragment processing are programmable stages from the graphics rendering pipeline and rasterization and output-merging are the stages that are non-programmable or configurable by using the GPU commands. In programmable stages, we need to write the program for vertices and fragments know as vertex shader and fragment shader respectively. Different programming models (C for graphics, high level shading language (HLSL), OpenGL Shading language (GLSL) etc.) have been introduced to write these shader programs [8]. In section IV, we have discussed how to program these shaders using GLSL and their perspective type of errors during writing code [5].

4. GLSL PROGRAM AND TYPES OF ERRORS

4.1 OpenGL Shading Language

The OpenGL Shading Language (GLSL) is the principle shading language for OpenGL. OpenGL provides many shading languages but GLSL is the one which is closer and the part of OpenGL core. GLSL is very popular programming model to write the shaders for GPU but it is very important to understand the graphics pipeline and the sequence of shader creating before writing in any language [1]. There are some particular steps to write a shader as follows:

- Create Shaders (creating new shader object)
- Specify Shaders (load shader source)
- Compiling Shaders (Actually compile source)
- Program setup (creating shaders object)
- Attach Shaders to Programs
- Pre-linking (setting some parameters)
- Link Shaders to Programs
- Cleanup (detach and delete all shader objects)
- Attach program to its stage
- Finally, let GPU know Shaders are ready.

4.2 Types of Error

During writing the vertex and fragment shaders by following above steps in GLSL there are variety of errors that occurs in a normal program. So, it is very important to have a deep look at those kind errors including how these errors occurs and reason of occurrence. These errors are as follows:

4.2.1. Error in Vertex Stream data

Error in vertex stream data is the type of error that normally occurs at very initial stage of writing the program when we try to get vertex buffer array (VBA). From GLSL side, a file name is passed that's buffer array is required. Before return the buffer array data of file there might be possibility of errors as follows:

- File doesn't exist at given path
- Accessed file is corrupted file
- Forgot to write method for reading the file

4.2.2. *Compilation Error*

On shader compilation, it must be checked that the compilation process of shader is successfully completed or not. Once compilation is completed, it returns the compilation status weather it is successfully compiled or not. In case of false there will be a compilation that should be handled from before moving to next stage.

4.2.3. *Missing vertex and fragment necessary parameters*

There are some necessary parameters for fragment and vertex that must be passed by accurate values in calling specific methods from GLSL:

glBindAttribLocation(vertexProgram , 0, "**Position**");
glBindFragDataLocation(geomFragProgram, 0, "**FragColor**");

Once these parameters are missed or wrong values are entered, there will be error occurred on compilation of these methods.

4.2.4. *Linking Error*

Similar to compilation, linking process can also be failed due to passing faulty programming for linking or some internal linking processing. There should be validation of linking process by returned value either its false or true mean linking process is successfully completed or not.

4.2.5. *Missing Program Attachment with Perspective Stage*

Another step during shader programming is attaching program with its perspective stage. In case, attachment process is skipped; there will be an error occurred in the program. Normally below methods is used for attachment.

glUseProgramStages(pipeline, VERTEX_BIT, vertxprogrm);

4.2.6. *Uniform Count Error*

In GLSL, a uniform is global variable that is declared with the keyword as "uniform" storage qualifier. These act as parameters that the user of a shader program can pass to that program. To pass the parameters following methods is used in GLSL:

glUniform2fv(lightposHandle, amount, lightpositions);

Here, vertex array length which should always be same as value as per array length. Dissimilar value will be cause of error in the program.

4.2.7. *Final Program Validation*

Finally, before using the program in GPU, the program validation is necessary. Program validation will tell us, the program is acceptable for GPU or not. Normally, this happen, a program is used for GPU without validation and due to error in program whole application crashed. Below method is used to validate the program:

glGetProgramiv(programme,GL_VALID_STATUS, &params);

Valid_Status parameter returns the status of program in true/false either the program is valid or not.

5. PROPOSED TESTING TOOL ARCHITECTURE

In last section we have discussed the number of challenges that might be occurred during writing the shader program using OpenGL Shading Language (GLSL). These errors in program will affect the system performance and efficiency. In order to avoid from these specific error, we have proposed testing tool architecture. By following this architecture, we can write an error free code for shading program to use in GPU. As in GLSL, there are number of steps to program a shader. We divided all the steps in four major categories as shown in figure3.

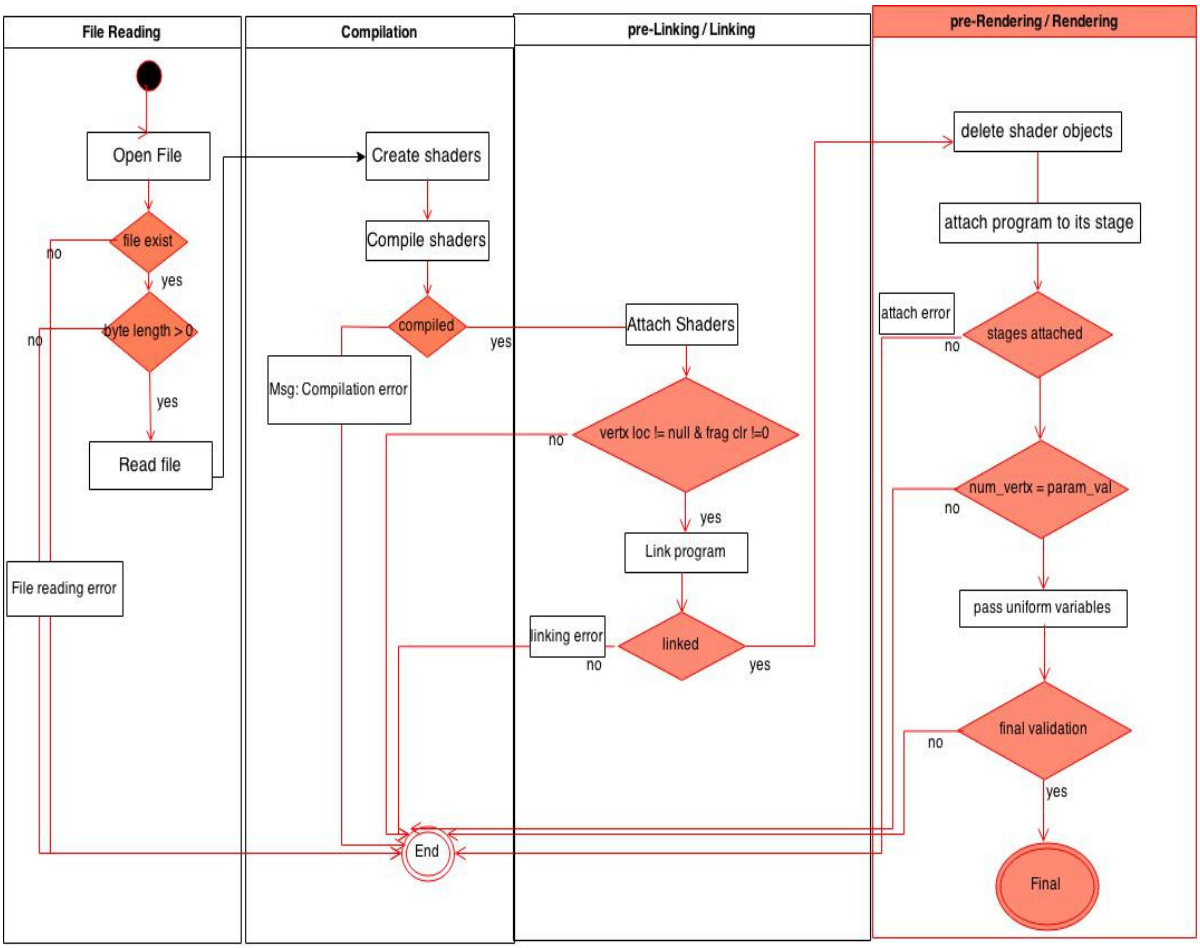

FIGURE 3: Testing Tool Architecture.

5.1. File Reading

The very first step is to read the file which's buffer array is required. In this section, our architecture validates some operations on that particular file to make sure that does the file exist at given path, does the file is valid and does the file read before passing to shader as a buffer array. If the file doesn't exist at give path or the accessing file is corrupted, the architecture imposes to terminate the program by writing the else cases and avoid to crash the program.

5.2. Compilation

In next section, after compilation it imposes to check the compilation results weather the compilation is completed successfully or not. In case of any error in compilation, the architecture enforces to terminate the program to avoid further crash in application.

5.3. Pre-Linking and Linking

After successful compilation, next step is validation of before and after linking the program. Before linking, some necessary correct parameters which are required for both vertex and fragment as well. The proposed architecture insists to pass the accurate required parameters such as position is required for vertex shader and colors for fragment. After linking, similar to compilation, it imposes to check that linking process is successfully completed or not and take action accordingly.

5.4. Rendering

Rendering step consist of many validations such as program attachment with its perspective stage in pipeline. According to our architecture, the programmer must write the GLSL statement which is used to attach the program with a specific stage. Conversely if this attachment is skipped, the program will unable to find its stage and application will be crashed.

Another type of error is related to uniform global variable as parameter value that passed for vertex specifically. glUniform2fv is method used that get three parameters in it. One of those is the length of vertex array which is created with uniform keyword. If the passing length value is dissimilar, error will be occurred in program and crashed. The last validation step in rendering section is validation of final program. Final program must be validated before using in GPU to make sure there is no error in the created program.

6. TESTING TOOL ARCHITECTURE EVALUATION

This testing tool architecture for OpenGL is proposed basically to improve exascale computing system. GPU is the basic unit that is being used to enhance the power of a system to achieve exascale computing. However, In order to achieve this certain level performance OpenGL play a major role to program graphical processing unit. This architecture helps us to detect the possible number of errors from the code written in OpenGL and improve the processing power of code as well. Keeping in view the program structure of an OpenGL Shading language, we have proposed the testing architecture to evaluation the number errors in our code that could be cause to decrease the performance of a system. Using this proposed testing tool architecture, we can evaluate our code written in GLSL and make error free by following it.

7. CONCLUSION

High performance computing architecture is the vision of next decade for exascale system. Many new technologies, algorithms and techniques are required to achieve exaflop computing power. In modern computers, GPU is basic unit that can provide the required performance for exascale system. So far, there are still many limitations for existing technologies as CUDA, OpenGL, and OpenCL etc. to program such a GPU unit. In order to enhance the performance of OpenGL code, we have presented a testing tool architecture in this paper. This proposed architecture insists the developers to write an accurate code by following the structure of proposed architecture. By future perspective, this proposed architecture will help to achieve high performance computing using OpenGL for exascale computing system. Still there are many open challenges for GPU in different technologies to develop an exascale computing system.

8. REFERENCES

[1]. J. Kessenich and D. Baldwin, “The OpenGL Shading Language”, Sep 2006 .

[2]. D. Luebke and G. Humphreys, “How GPU works”. Feb 2007.

[3]. M. J. Kilgard and J. Bolz, “GPU-accelerated Path Rendering”, Computer Graphics Proceedings, Annual Conference Series. 2012.

[4]. “Exascale computing research.” Internet: http://www.exascale-computing.eu/presentation-2/, [May. 3, 2015].

[5]. “3D Graphics with OpenGL Basic Theory” http://www.ntu.edu.sg/home/ehchua/programming/opengl/ cg_basicstheory.html, July. 2012 [April, 14. 2015].

[6]. “GLSL Tutorial Core” http://www.lighthouse3d.com/opengl/glsl/, June, 23. 2014 [May. 10, 2015].

[7]. D. Shreiner, “Performance OpenGL: Platform Independent Techniques”, SIGGRAPH. 2001.

[8]. S.F. Hsiao, P. Wu, C.W. Sheng, and L.Y. Chen, “Design of a Programmable Vertex Processor in OpenGL ES 2.0 Mobile Graphics Processing Units”. IEEE. 2013.