PROCEEDINGS OF SCIENCE

Designing A Unified Architecture Graphics Processing Unit

Lingjuan Wu[1]
Wuhan Digital Engineering Institute
Wuhan, 430205, China
E-mail: wljuan17503@163.com

Liang Huang[2]
Wuhan Digital Engineering Institute
Wuhan, 430205, China
E-mail: hvb60@163.com

Tinggang Xiong[3]
Wuhan Digital Engineering Institute
Wuhan, 430205, China
E-mail: xtg_hb@aliyun.com

Graphics Processing Unit (GPU) performs graphics computing and its architecture has developed from the fixed function pipeline to the programmable unified pipeline. Unified architecture promises dynamic load balancing and guarantees the high parallel computing of GPU. This paper presents the design and implementation of a unified architecture GPU. The unified shader is based on the SIMD and SIMT architecture. On the thread level, SIMT guarantees the full-load capability of unified shader by thread managing and scheduling. On the instruction level, SIMD controls the execution of the unified shader hardware unit. We finish the algorithm, architecture design and Verilog RTL implementation. The verification results on FPGA show that the proposed GPU works correctly and its vertex and fragment processing speed reaches one unit per clock cycle.

CENet2017
22-23 July, 2017
Shanghai, China

[1]Speaker
[2]This study is supported by National Natural Science Foundation of China (61403350)
[3] Corresponding Author

PoS(CENet2017)079

1. Introduction

Graphics Processing Unit is a highly parallelled processor specialized for 2D and 3D graphics computing. With the increasing demand for higher performance and higher resolution graphics applications, GPU becomes a key factor in computer embedded system design. Modern GPU is programmable and has its own instruction set like CPU, but with much more powerful parallelled computation capability. GPU can also be used for general propose computing known as GPGPU [1-3].

The concept of GPU was proposed by Nvidia in GeForce 256 in 1999 which was based on the fixed function pipeline. It is the first time that geometry transformation, lighting, and texture mapping are implemented by hardware while before that graphics computing were implemented by CPU. Hardware implementation improves the computing speed but lacks of flexibility because these hardware units are not programmable.

Since then, the programmable pipeline [4] and unified graphics pipeline [5-7] have been introduced to GPU design for higher performance and programmability. In programmable pipeline, vertex shader and fragment shader are introduced for vertex and fragment computing respectively. And the shader is programmed by the GLSI language on user application level. This architecture improves the GPU's programmability, but vertex and fragment are computed in separate hardware modules. For programs with more vertex than fragment, the vertex shader works in full load but the fragment shader is idle and vice versa. Thus, unified graphics pipeline is introduced.

In the unified graphics pipeline, a hardware unit called unified shader executes vertex, fragment and geometry programs [8]. The data flow is shown in Figure1. GPU accepts data and commands from the CPU.Vertices are firstly processed in the unified shader for geometry transformation, lighting computation and texture coordination calculation. Then after primitive assembly and rasterization, fragments are generated for each primitive. Fragments are further processed in the unified shader for lighting and fog processing. Finally, pixels are generated in pixel engine after antialiasing, scissor test and stencil test. One fragment generates one pixel or several fragments are interpolated to generate one pixel based on antialiasing algorithm. Pixels are stored in the frame buffer and will be displayed on the monitor. Unified architecture promises dynamic load balancing of shader and improves hardware utilization.

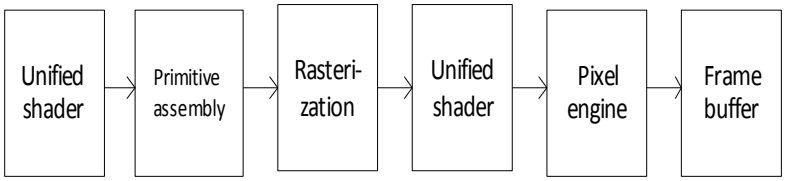

Figure 1: Unified Graphics Pipeline

In this paper, we present the design and implementation of a unified architecture GPU based on SIMT (single instruction multiple thread) and SIMD (single instruction multiple data) shader unit. SIMT implements thread scheduling and management on the thread level and makes sure that the shader unit works in full load state. SIMD manages the hardware unit with instruction scheduling. The unified shader is designed based on extendable processing element and the computing capability can be improved by integrating more elements. We finish the algorithm, architecture and Verilog HDL design. The verification results on Xilinx FPGA show

that the proposed GPU works correctly, and vertex and fragment processing speed reaches one unit per clock cycle respectively.

The rest of this article are organized as follows. In Section 2, we describe the GPU's architecture and the design of the unified shader, rendering engine and texture engine. Experiment results of the proposed GPU on Xilinx FPGA and SMIC 40nm technology are presented in Section 3. We summarize the paper with a conclusion in Section 4.

2. Unified Architecture GPU Design

2.1 Unified Shader

The block diagram of the unified architecture GPU designed is shown in Figure 2. It mainly includes command processor, unified shader, rendering engine, texture engine and pixel engine. The communication and synchronization between each module is based on the valid-ready protocol. The data buffer is designed for each module to improve the throughput. The whole graphics computing pipeline is compatible with single-precision floating point IEEE 754 standard to guarantee the precision. The asynchronous FIFO is used for signals cross the clock domain. In this article, we focus on the design of the unified shader, rendering engine and texture engine because they are the key modules in graphics processing pipeline.

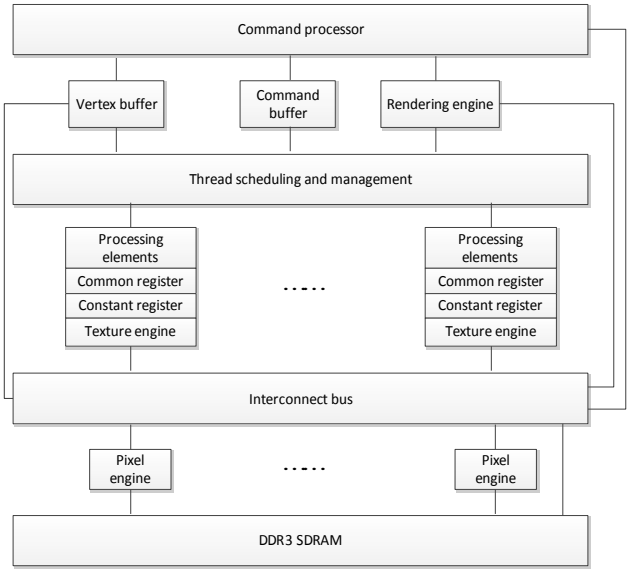

Figure 2: Unified Architecture GPU

Command processor controls the whole graphics pipeline. Application program written in the graphics API such as OpenGL is analysed and compiled to generate GPU command of the driver. The command is stored in the command buffer, and vertex information is stored in the vertex buffer. When the graphics pipeline starts to execute, data and command are read into GPU and further analysed to control the whole pipeline by the command processor.

Unified shader executes vertex and fragment shading programs, and designates the processing unit dynamically to secure the load balancing. In our design, the unified shader is based on SIMT combined with SIMD architecture to make the best use of the hardware units. On the thread level, SIMTOn the instruction level, SIMD controls the execution of the unified shader hardware unit.

The unified shader consists of 128 hardware units called processing element (PE). Four PE units constitute one SIMD unit, and eight SIMD units form one block processor. In this paper, we take the GPU with four block processors as an example, though the GPU can definitely integrate more block processors for higher computing capability. Each block processor can execute 2048 threads in parallel, and these threads are divided into 64 groups with each group including 32 threads. The thread scheduling and management module controls the thread execution. For example, when one thread is stalled, its state information will be stored and another thread will start to execute.

In SIMT design, threads are scheduled and managed by group. The context information for each thread group is saved during scheduling and mainly includes group number, common registeration starting address and program address. There are 64 thread groups and the block processor chooses one group to execute. The thread group assigned earlier has higher priority. And one instruction execution takes four clock cycles. The 32 threads in one group execute the same instruction stream from the same address. The instruction is vertex or fragment processing program. But during execution each thread runs independently with its specific registeration space. The block processor reaches the maximum performance when all the threads in one group have the same path.

On the instruction level, thread is executed by SIMD, and four hardware units execute the same instruction. In graphics processing, the attribute of vertex or fragment contains four components which are computed by the four hardware units in SIMD. For example, the component of position is XYZW, then the color is RGBA.

Based on the architecture and algorithm described above, we finish the unified shader RTL design. The unified shader is programmable and includes a four-stage pipeline: IF, ID, EX, WB. And its instruction set includes forty instructions and mainly includes arithmetic, control, lighting and texture mapping. Each instruction is 128 bit and the operation code is 5 bit. Hardware module is mainly composed of instruction fetch unit, decoder, address generator, operand fetch unit, registeration, ALU, control unit and output registeration. The instruction fetch unit read vertex and fragment processing commands from memory. After instruction decoding, source and destination operand address are generated in the address generator. Then source operand fetch unit read data from constant, temporary or input register. According to the operation code, ALU performs arithmetic operation such as adding, multiplying and multiply-adding. Transcendental function unit (TFU) carries out complex mathematics computation such as trigonometric function, exponential, reciprocal and etc. TFU is designed based on the homogeneous polynomial approximation and look-up table algorithm. Control unit implements call, branch or loop operation. Finally, the output results are written back to the output registeration.

2.2 Rendering Engine

Rendering engine accepts vertices from the unified shader, generating fragments within each primitive and computing the attribute of each fragment. It mainly includes primitive assembly, clip, setup, fragment generation and interpolation as shown in Figure 3. In the design, primitive type of point, line list, line strip, triangle list, triangle strip and triangle fan are supported.

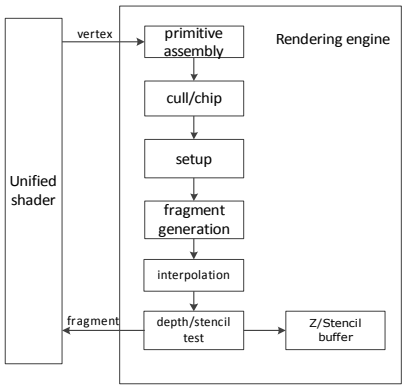

Figure 3: Rendering Engine Design

Graphics scenes are built based on the basic primitives, and each scene contains numerous primitives. Each primitive consists of many fragments depending on its geometry size. Therefore, the computation capability and speed of rendering engine are two key factors in the whole graphics pipeline. In order to achieve the real-time graphics computing, we need to compromise between algorithm and hardware complexity.

Primitive assembly module assembles vertices into primitive depending on the primitive type. For example, two vertices are assembled to a line and three vertices are assembled to a triangle. After assembling, the basic unit is primitive in the following graphics pipeline. Culling and clipping are performed for each primitive. First, primitives that are outside of the view frustum are culled. And then the remaining primitives with vertex outside of the frustum need to be clipped. During clipping, new vertices are generated on the boundary of the view frustum and then new primitives are built.

Cohen-Sutherland algorithm is explored to evaluate the geometrical relationship between each primitive and the view frustum [9]. Since the vertex's coordinate is a four dimensional homogeneous coordinate *(x,y,z,w)* in the clip space, the view frustum is defined by the fourth coordinate *w*. We define 6-bit region code to represent the comparison result of {z>w, z<-w， y>w, y<-w, x>w, x<-w}, the vertex is outside of the corresponding boundary when the result is 1. On the contrary, the vertex is inside the boundary when the result is 0. For example, 000000 means the vertex is inside the view frustum and 100000 means the vertex is outside of the far-z boundary.

Based on the region code, we take triangle as an exampl. If its three vertices are all outside the view frustum, the triangle should be culled, and if one or two vertices are outside of the view frustum, the triangle should be clipped and new vertex will be generated to from a new triangle. The equation for calculating the new vertex's attribute is

$$C = tC_0 + (1-t)C_1 \tag{2.1}$$

where *C* represents the component of each attribute, for example, the four components of color *RGBA*. $C_0 C_1$ is the component of the two vertices which form one edge of the triangle that intersects with the boundary. After primitive assembly and clipping, the coordinate of each vertex in the primitive is transformed to the normalized device coordinate and finally to the 2D window coordinate.

The setup module makes preparation for rasterization by calculating the initial point and direction. It mainly includes control and data path module. As the vertex coordinate is in a 2D window space, clipping is further performed for each primitive based on the resolution

information such as 1920x1080. Primitives that have vertex outside of the window are clipped to further confine the area of rasterization.

The rasterization module calculates the attribute of each fragment and mainly includes fragment generation and interpolation. First of all, we scan the fragment from the initial point, and then use edge equation to check whether the fragment is inside the primitive. If the fragment is inside, we calculate the fragment's attribute, otherwise we move to the next fragment in the rasterization direction. Linear interpolation algorithm is explored to compute each fragment's arrtibute.

Each fragment may have 12 attributes at maximum and each attribute has 4 components. Thus, parallelled hardware computation is explored in our design to improve the speed as shown in Figure 4. Primitives are classified as odd and even and rasterized in parallel. Furthermore, the depth and stencil test is performed for each fragment to delete the invalid fragment in early stage.

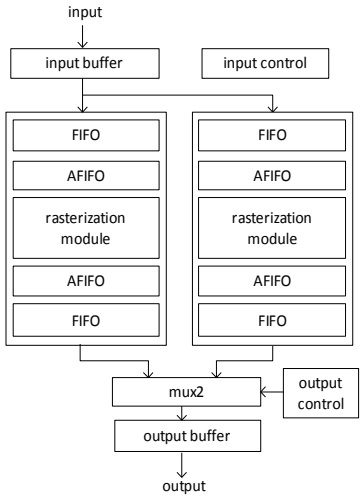

Figure 4: Parallel Rsterization Module

2.3 Texture Engine

Texture engine accepts fragments from the unified shader and performs texture mapping for each fragment. The texture engine we designed mainly includes four components as shown in Figure 5: controller, texture address generator, format converter and texture cache. There are sixteen texture engine units working in parallel in GPU.

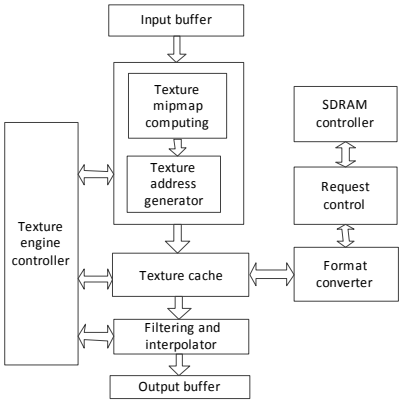

Figure 5: Texture Engine Design

Controller orchestrates the texture engine. As we mentioned above, each fragment may have 12 attributes and one attribute is texture coordinate (s,t,r,q). The address generator module calculates the address of the corresponding texture data in SDRAM from (s,t,r,q). The texture mapping area is not always the same as the texture in SDRAM. For example, if the mapping area is much bigger, up-sampling is needed, otherwise, down-sampling is needed. Therefore, mipmap technique is explored, in which textures of different sizes are stored in SDRAM and the one which has the approximate area with the mapping area is used to improve speed and precision. In our design, 13 level mipmap is supported and the maximum texture resolution is 4kx4k.

The texture data of different materials is stored in SDRAM and compressed to save memory. In texture mapping, texture data read out from SDRAM are converted to ARGB8888 data format. The format converter module supports various compressed format such as ARGB1555, YUY2, YV12 and etc.

A cache is designed to save texture data read out from SDRAM after format converting. The cache size is 2KB and contains 32 cache lines. Considering the locality characteristic of the texture data, a tile-based method is explored in cache design. Each tile contains a 4x4 texel, where texel is the unit of texture data and is 32bit. In our design, each cache line is 64Byte, and can store one tile. Furthermore, each cache line is divided into four banks. Each bank is 128bit, and in turn can store 4 texel, as shown in Figure 6. Thus, if the texture coordinate is located among four texels in one tile, we can read out the four texel data at the same time. The reason is that each bank can be read and written separately. Otherwise, if the texture's coordinate is located among different tiles, different cache lines are read to get the texel data. Finally, the texel data read out from the cache is interpolated and filtered to generate the texture data for each fragment.

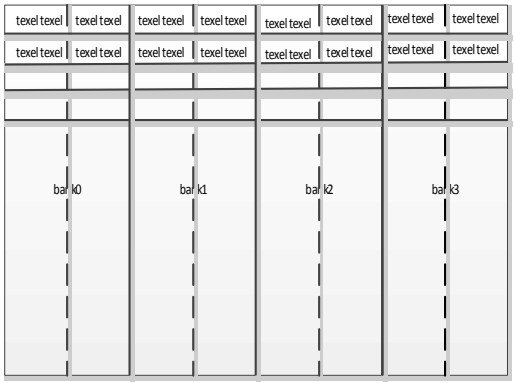

Figure 6: Texture Cache Design

3. Experiment Results

We finished the design of the unified architecture GPU with the top-down methodology. Base on the architecture and algorithm described, we finished the GPU Verilog RTL design and and developed the OpenGL 2.0 subsystem driver. The proposed GPU was implemented on Xilinx Vertex7 FPGA to evaluate its performance.

Various OpenGL 2.0 programs are tested. The hardware verification platform of GPU and some of the results are shown in Figure7. In Figure7(a) and Figure7(b), GPU draws a cube and gears respectively, and the object rotates when viewport is changed by pressing specific buttons

on the keyboard. In Figure 7(c), GPU draws a cube and does texture mapping on each face of the cube. We also run a game for comprehensive testing of GPU as shown in Figure7(d). The results show that the proposed GPU performes 3D graphics computing correctly.

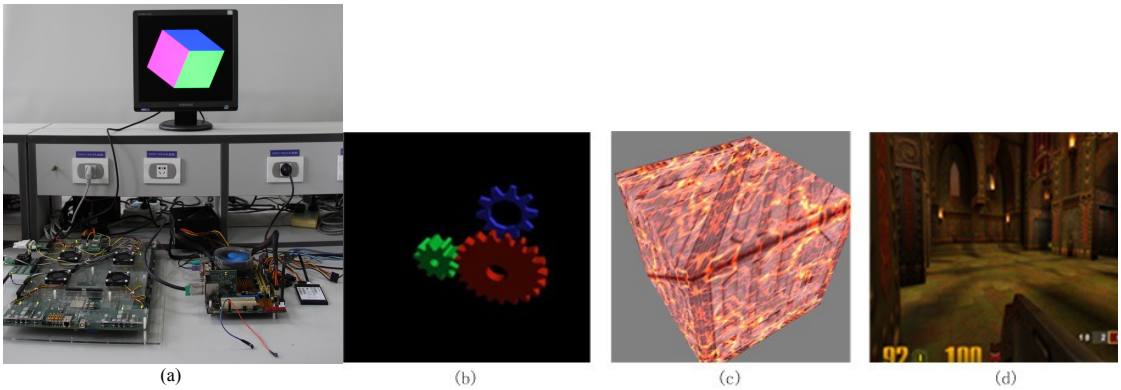

Figure 7: Verification of GPU on FPGA

Furthermore, we use Vivado to collect the GPU hardware signal. As shown in Figure 8 signal cycle[3:0] represents the thread number while each thread is one program for vertex or fragment processing. The value of cycle [3:0] switches per clock cycle, which means GPU has the capability of processing one vertex or fragment per clock cycle.

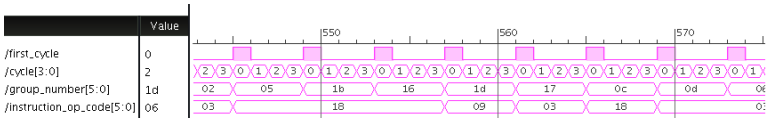

Figure 8: GPU Hardware Simulation Result

We further synthesize the Verilog HDL code under the SMIC 40nm technology node. The synthesis results show that the core frequency of the proposed GPU reaches 600MHz, which means that GPU vertex or fragment processing speed is 600M/s. We also finish the layout design, and the silicon area is about 7.5mm*7.5mm.

To the best of our knowledge, the unified architecture is the mainstream in the GPU design. Sun designed a unified architecture shader GPU with its core frequency and vertex processing speed 400M, 300M/s [10]. And the core frequency and vertex speed is 200M, 50M/s in the GPU Sohn proposed [11]. In industry, Nvidia adopts SIMT architecture and the core frequency reaches 1178M in Geforce GTX 960. And AMD adopts SIMD architecture and the core frequency reaches 1050M in Radeon R9. Commercial GPU products improve the speed by integrating the numerous computing units.

The GPU we proposed is based on SIMT combined with SIMD architecture. And the speed reaches 600M. There are 128 PE units in the unified shader, 16 texture engine units and 8 pixel engine units. More hardware units can be integrated in the GPU to improve the speed. In conclusion, we expect higher speed when more computing units are integrated in the proposed GPU.

4. Conclusion and Future Work

In this paper, we present the design and implementation of a unified architecture GU. And we mainly introduce the designing of the unified shader, rendering engine and texture engine because they are the key modules in the graphics pipeline. The GPU is designed with the top-

down methodology and Verilog HDL code is provided. The verification results on FPGA show that the proposed GPU works correctly, and the vertex and fragment processing speed reaches one unit per clock cycle. In the future, more computing units will be integrated in the GPU to improve computing capability and speed.

References

[1] C.J.Thompson, S.Hahm, M.Oskin. *Using modern graphics architecture for general-purpose computing: a framework and analysis*[C], IEEE/ACM International Symposium on Microarchitecture, 2002:306-317

[2] J.D.Owens, D.Luebke, N.Govindaraju,M.Harris, J.Kruge. *A survey of general-purpose computation on graphics hardware*[J], Computer Graphics Forum, 2007, 26(1):80-113

[3] T.D.Han, T.S.Abdelrahman. *hiCUDA: high-level GPGPU programming*[J], IEEE Transactions on Parallel and Distributed Systems, 2011, 22,(1):78-90

[4] V.M.Barrio, C.Gonzalez, J.Roca, A.Fernandez. *ATTILA: a cycle-level execution-driven simulation for modern GPU architecture*, in Proc. International Symposium on Performance Analysis of System and Software[C], 2006:231-241

[5] E.Lindholm, J.Nickolls, S.Oberman, J.Montrym. *Nvidia Tesla: a unified graphics and computing architecture*[C], IEEE Micro, 2008:39-55

[6] V.Moya, C.Gonzalez, J.Roca, A.Fernandez, R.Espasa. *Shader performance analysis on a modern GPU architecture*[C], MICRO2005

[7] A.Maashri, G.Sun, X.Dong, V.Narayanan, Y.Xie. *3D GPU architecture using cache stacking: performance, cost, power and thermal analysis*[C], in Proc.International Conference on Computer Design (ICCD), 2009

[8] J.Han, L.Jiang, H.Du, X.Cao, L.Dong, L.Meng. *Hardware accelerator and 3D pixel shader architecture for computer graphics*[J], Journal of Computer-aided Design Computer Graphics, 2010, 22(3) :363-372

[9] B.Jiang, J.Han. *Improvement in the Cohen-Sutherland line segment clipping algorithm*[C], IEEE International Conference on Granular Computing, 2013:157-161

[10] G.Sun. *Design and research of unified atchitecture shader based on automatic threading and vliw*[D], Hangzhou: Zhejiang Univeristy, 2012 (In Chinese)

[11] J.Sohn, J.Woo, M.Lee, H.Kim, *A 155mw 50-mvertices/s graphics processor with fixed-point programmable vertex shader for mobile applications*[J], IEEE Journal of Solid-State Circuits, 2006, 41(5):1081-1091