International Journal of Engineering, Management, Humanities and Social Sciences Paradigms (IJEMHS)
Volume 30, Issue 04, Quarter 04 (Oct-Nov-Dec 2018)
ISSN (Online): 2347-601X
www.ijemhs.com

ATTILA: A Simulator for Modern GPU Architectures

Vidya Mohanty
Aryan Institute of Engineering & Technology, Bhubaneswar

Abstract

The present work presents a cycle-level execution-driven simulator for modern GPU architectures. We discuss the simulation model used for our GPU simulator, based in the concept of boxes and signals, and the relation between the timing simulator and the functional emulator. The simulation model we use helps to increase the accuracy and reduce the number of errors in the timing simulator while allowing for an easy extensibility of the simulated GPU architecture. We also introduce the OpenGL framework used to feed the simulator with traces from real applications (UT2004, Doom3) and a performance debugging tool (Signal Trace Visualizer). The presented ATTILA simulator supports the simulation of a whole range of GPU configurations and architectures, from the embedded segment to the high end PC segment, supporting both the unified and non unified shader architectural models.

1. Introduction

We have developed a generic GPU microarchitecture containing most of the advanced hardware features seen in today's major GPUs. We have liberally blended techniques from all major vendors and the research literature [26], producing a microarchitecture that closely tracks today's GPUs without being an exact replica of any particular product available or announced. We have then implemented this microarchitecture in full detail in a cycle-level, execution-driven simulator. In order to feed this simulator, we have implemented an OpenGL framework comprised by a library, a driver and a capture tool. The OpenGL framework allows to run traces from modern graphic applications (i.e. games) like UT2004 and Doom3 in our simulator. Our microarchitecture and simulator are versatile and highly configurable and can be used to evaluate multiple configurations: high-end PC GPUs [1] to embedded GPUs [2] for mobile systems.

The remainder of this paper is organized as follows: Section 2 describes the rendering algorithm and the microarchitecture of a GPU as implemented by the ATTILA simulator. Section 3 discusses the simulation model and the structure of the different simulator components. Section 4 introduces our OpenGL framework, used to feed the simulator with traces from real graphic applications. In section 5 a simple experimental test case is presented. Finally sections 6 and 7 present related work, conclusions and future work.

2. ATTILA Architecture

The 3D Rendering Algorithm

The 3D rendering algorithm can be defined using the stream programming model [11] in terms of streams and kernels. The input stream is a list of vertices and their input properties (position, color, texture coordinates), named attributes in

The rendering algorithm implemented in modern GPUs is based on the rasterization of shaded polygons on a color buffer, using a Z buffer to solve the visibility problem. GPUs are based on the rasterization of triangles because the simplicity and efficiency of hardware triangle rasterizers. Therefore all surfaces forming the scene to render are transformed into triangles (tesselation) in an offline preprocess. Coplanar four vertex polygons, named quads in OpenGL, are supported as two triangles. Some GPUs also support the tesselation of high order surfaces (Bezier, N-Patches) using specific hardware.

The rendered image, stored in the framebuffer, a 2D matrix array, contains the properties of all the visible parts in the rendered scene from the view point of a defined observer. In most cases the stored property is the surface color. The properties of the rendered surfaces are calculated at two points: at the vertices of the triangles that form the surface and at the fragments generated by the rasterization of those triangles. Early graphic processors performed most of the computation, mostly related to the illumination from a number of light sources, at the vertex level with the fragment properties being linearly interpolated from the vertex properties (Goraud shading). With the modern GPUs high fragment processing power most of those computations have moved to the fragment level (Phong shading). At the vertex level remain the transformations related to geometry and physics.

OpenGL. The vertex stream, named batch in OpenGL, can be indexed to enable reusing the computation of vertices from adjacent triangles. The input vertex stream is fed into a shader kernel executing the vertex shader: a program that transforms

(coordinate system convertion, lighting, etc.) the properties of the input vertices. The transformed vertex stream is feed into a kernel that assembles vertices as triangles. This triangle stream passes through geometry related kernels (clipping, face culling, triangle setup) that generate or remove triangles and prepare the triangle stream to be processed by the rasterizer. The rasterizer or fragment generator kernel pieces the triangles into small fragments equivalent to a pixel (an element in the framebuffer) with fragment properties copied or linearly interpolated from the input triangle vertex properties.

The fragment stream is then processed by a number of fragment kernels that remove non visible fragments and compute the final fragment attributes. The fragment test kernels usually implemented in the rendering algorithm are: scissor test, alpha test, stencil test and z test . Scissor removes triangles outside a defined rectangle window. Alpha removes transparent fragments based on a defined constant and the fragment color alpha component. Stencil removes fragments based on a per pixel mask, the stencil buffer. The stencil test is performed comparing a defined reference value against the value stored for the corresponding fragment pixel. The per pixel stencil value is optionally updated based on the result of the stencil and z tests applying different update functions (increment, decrement, etc.). The depth (z) test compares the fragment depth against the depth value of the last fragment drawn over the pixel, as stored in the depth (z) buffer. Depending on the selected compare function fragments behind (smaller z), ahead (larger z) or at the same depth (equal z) are allowed to flow to the next procesing kernels or discarded.

Final fragment properties are computed by a shader kernel similar to the vertex shader kernel. Fragment shaders (and in recent implementations vertex shaders) are allowed to use non-streaming data, reading from one, two or three dimensional buffers named textures, for their computations.

The processing order of the test and fragment shader kernels isn't fixed. Scissor test can be performed before shading, alpha test must be performed after shading. Stencil and depth can be performed before shading if the fragment depth isn't modified by the fragment shader and alpha test is disabled.

3. Conclusions

We have presented an highly configurable simulator for a modern GPU architecture that is implemented using the box and signal simulation model. The simulated architecture implements the unified shader architecture that will be present in future GPUs. We have developed an OpenGL framework that allows to capture and simulate trace from real graphic applications, games, as Doom3 and UT2004. The simulator generates a large amount of statistic data and data flow information that can be used to evaluate different microarchitecture implementations for all the pipeline stages.

We will increase OpenGL framework to support more games. We are also working on a backend for the glSlang compiler. In the future we will start a Direct3D framework. We plan to upgrade the shader to Shader Model 3.0 and glSlang functionality level implementing branching and predication. We will implement additional features from modern GPUs: texture compression methods [25], render to texture, floating point buffers and textures; color compression; double rate Z and stencil; double sided stencil; supersampling and multisampling based antialiasing [23].

References

[1] Victor Moya, Carlos Gonzalez, Jordi Roca, et al. Shader Performance Analysis on a Modern GPU Architecture. Micro 38, 2005.
[2] Victor Moya, Carlos Gonzalez, Jordi Roca, et al. A Single (Unified) Shader GPU Microarchitecture for Embedded Systems. HiPEAC 2005.
[3] Erik Lindholm, et al. An User Programmable Vertex Engine. ACM SIGGRAPH 2001.
[4] WO02103638: Programmable Pixel Shading Architecture, December 27, 2002, NVIDIA CORP.
[5] Beyond3D Graphic Hardware and Technical Forums. http://www.beyond3d.com
[6] DIRECTXDEV mail list. http://discuss.microsoft.com/archives/directxdev.html
[7] T. Aila, V. Miettinen and P. Nordlund. Delay streams for graphics hardware. ACM Transactions on Graphics, 2003.
[8] T. Akenine-Möller and J. Ström Graphics for the masses: a hardware rasterization architecture for mobile phones. ACM Transaction on Graphics, 2003.
[9] Stanford University GLSim & GLTrace. http://graphics.stanford.edu/courses/cs448a-01-fall/glsim.html
[10] J. W. Sheaffer, et al. A Flexible Simulation Framework for Graphics Architectures. Graphics Hardware 2004.
[11] J. Owens, B. Khailany, et ak. Comparing Reyes and OpenGL on a Stream Architecture. Graphics Hardware 2002.
[12] T. J. Purcell, I. Buck, W. R. Mark, P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, 2002.
[13] Greg Humphreys, Mike Houston, Ren Ng. Chromium: A Stream Processing Framework for Interactive Rendering on Clusters. SighGraph 2002.
[14] Marc Olano, Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. Graphics Hardware, 2000.
[15] Michael D. McCool, et al. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. Proceedings Graphics Hardware 2001.
[16] J. McCorkmack, et al. Neon: A (Big) (Fast) Single-Chip 3D Workstation Graphics Accelerator. WRL Research report 1998.
[17] Green, N. et al. Hierarchical Z-Buffer Visibility. Proceedings of SIGGRAPH 1993.
[18] S. Morein. ATI Radeon Hyper-z Technology. In Hot3D Proceedings - Graphics Hardware Workshop, 2000.
[19] US20030038803: System, Method, and apparatus for compression of video data using offset values. ATI Technologies.
[20] Ziyad S. Hakura, Anoop Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. ISCA 1997.
[21] Homan Igehy, et al. Prefetching in a Texture Cache Architecture. Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware
[22] Se-Jeong Park et al. A reconfigurable multilevel parallel texture cache memory with 75-GB/s parallel cache replacement bandwidth. Solid-State Circuits, IEEE Journal of May 2002.
[23] Liu Ren, et al Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. EUROGRAPHICS 2002.
[24] EXT_texture_compression_s3tc. http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt
[25] Simon Fenneyy. Texture Compression using Low-Frequency Signal Modulation. Graphics Hardware (2003).
[26] Stanford University CS488a Fall 2001 Real-Time Graphics Architecture. Kurt Akeley, Path Hanrahan.
[27] Lars Ivar Igesund, Mads Henrik Stavang. Fixed function pipeline using vertex programs. November 22. 2002
[28] Joel Emer, et al. Asim: A Performance Model Framework. IEEE Computer, February 2002 (Vol. 35, No. 2).
[29] Microsoft Meltdown 2003, DirectX Next Slides. http://www.microsoft.com/downloads/details.aspx?FamilyId=3319E8DA-6438-4F05-8B3D-B51083DC25E6&displaylang=en
[30] ARB Vertex Program extension: http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt
[31] ARB Fragment Program extension: http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.
[32] GPGPU http://www.gpgpu.org/
[33] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. Graphics Hardware 2004.
[34] Daniel Horn, Mike Houston, and Pat Hanrahan. ClawHMMer: A Streaming HMMer-Search Implementation. Supercomputing 2005.
[35] GPUBench: http://graphics.stanford.edu/projects/gpubench/
[36] Jeremy W. Sheaffer, Kevin Skadron, David P. Luebke. Studying Thermal Management for Graphics-Processor Architectures. ISPASS 2005.
[37] Mesa 3D Graphics Library. http://www.mesa3d.org/