ATTILA: A Cycle-Level Execution-Driven Simulator for Modern GPU Architectures

Victor Moya del Barrio, Carlos González, Jordi Roca, Agustín Fernández
Department of Computer Architecture, Universitat Politècnica de Catalunya
{vmoya, cgonzale, jroca, agustin}@ac.upc.edu
Roger Espasa
Intel, DEG, Barcelona
roger.espasa@intel.com

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

2.1 The 3D Rendering Algorithm

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.

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 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.

The last fragment kernel uses the fragment properties (usually the color) computed by the fragment shader kernel to update the framebuffer, either overwriting the pixel or using more complex blending functions.

Modern GPUs implement the described algorithm as a very large multistaged pipeline implementing each kernel in hardware. Modifications and improvements to the algorithm, for example Hierarchical Z or occlusion tests, are implemented to accelerate the rendering. The GPU's hardware interface is closed and graphic applications must use standardized graphic APIs for which the GPU vendors provide their own implementations. Direct3D (defined by Microsoft) and OpenGL (defined by a standardization board of software and hardware vendors) are the APIs implemented by all vendors.

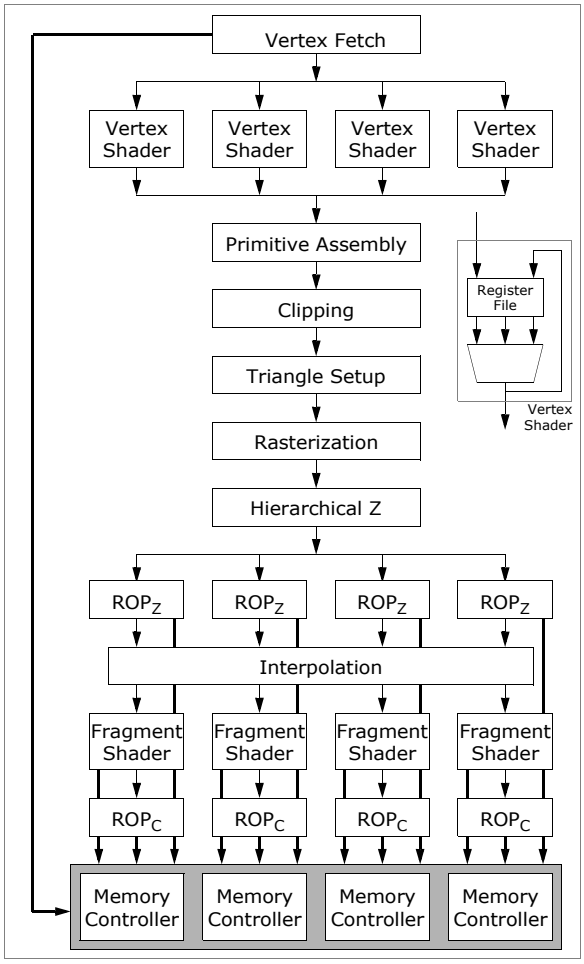

Figure 1. ATTILA Non-Unified Architecture.

2.2 Detailed ATTILA GPU Pipeline

This section describes our implementation of the rendering pipeline in hardware. We have blended techniques and ideas from different vendors and publications and we have made educated guesses in those areas where information was specially scarce. Our implementation correlates in most aspects with current GPUs, except for one design decision: we decided to support from the start a unified shader model [29] in our microarchitecture. Although the simulator can be configured to emulate today's hard partitioning of vertex and fragment shaders, the microarchitecture is thought out as having a unified shader model.

Figure 1 shows a block diagram of the ATTILA GPU pipeline configured with separated vertex and fragment shaders. Figure 2 shows the same pipeline configured with a unified pool of shaders. The input and output processing elements of the different ATTILA units can be found in Table 1, as well as their latencies in cycles, bandwidths and queue sizes. The table shows a baseline configuration implementing four vertex shaders (non-unified), two fragment (or unified) shaders each processing 4 fragments per cycle, two fragment test and framebuffer update units each processing 4 fragments per cycle, four channels to GPU memory, each providing a bandwidth of 16 bytes per cycle, and two buses with system memory, providing a bandwidth of 8 bytes per cycle (not shown in the figures). Both pipelines show the datapath for z and stencil performed before shading. The alternate datapath, after shading, is implemented but not shown.

Unit	Input Bandwidth	Output Bandwidth	Input Queue		Latency
			Size	Element width	
Streamer	1 index	1 vertex	48	16×4×32	Mem
Primitive Assembly	1 vertex	1 triang.	8	3×16×4×32	1
Clipping	1 triang.	1 triang	4	3×4×32	6
Triangle Setup	1 triang.	1 triang	12	3×4×32	10
Fragment Generation	1 triang	2×64 frag.	16	3×4×32	1
Hierarchical Z	2×64 frag.	2×64 frag.	64	(2×16+4×32)×4	1
Z Test	4 frag.	4 frag.	64	(2×16+4×32)×4	2+Mem
Interpolator	2×4 frag.	2×4 frag.	-	-	2 to 8
Color Write	4 frag.		64	(2×16+4×32)×4	2+Mem
Vertex Shader	1 vertex	1 vertex	12+4	16×4×32	variable
Fragment Shader	4 frag.	4 frag.	112+16	10×4×32	variable

Table 1: Bandwidth, queues and latencies in cycles for the baseline ATTILA architecture

The Command Processor (not shown in Figure 1) is the unit that controls the whole pipeline, receiving and processing the commands sent by the system CPU. The Command Processor's tasks are to control the rendering of batches and handle buffer writes (textures, vertex and index buffers) from system memory to GPU memory. Our current implementation allows to pipeline render state changes and buffer writes concurrently with rendering a batch. Batch rendering is divided in two phases: geometry phase (up to the Clipper stage) and fragment phase (starting from the Triangle Setup stage) to allow the pipelining two batches (one in the fragment phase and one in the geometry phase).

The Streamer unit task is to request input vertex attribute data to the Memory Controller, convert the data to the internal format (4 component 32 bit float point vectors) and issue vertices to a shader unit. A vertex post shading cache, storing indexed vertices already shaded vertices, enables reusing the vertex shader results for vertices in adjacent triangles.

The Primitive Assembly stage stores vertices and assembles them as triangles. We support five OpenGL primitives: triangle lists, fans and strips and quad lists and strips.

The Clipper unit clips the triangles against the view frustum volume. Our current ATTILA implementation is limited to perform trivial rejection of those triangles that lay completely outside the volume. All other triangles, including partially included triangles, flow free to the Rasterizer units.

The implemented rasterization algorithm is based on the 2D Homogeneous rasterization algorithm described by Olano and Greer [14]. Homogeneous rasterization removes the need of implementing clipping as division by w is no longer required when creating the triangle edge equations. Still, we divide by w the triangle vertex positions, except for triangles with w = 0, to generate a starting point for the triangle scan algorithm and to calculate the triangle bounding box [14]. Triangle Setup calculates the triangle half-plane edge and a depth (z/w) interpolation equations from the triangle homogeneous matrix. The equation coefficients calculated are then fed to the Fragment Generator unit.

The Fragment Generator traverses the triangle area projected in the viewport and iteratively generates fragments. Generated fragment have the following attributes: a 2D coordinate, the triangle's three edge equations values (used for the inside triangle test and as barycentric coordinates for attribute interpolation), a cull flag (outside viewport or triangle) and the fragment depth (used the Hierarchical Z test and Z test).

Our current Fragment Generator supports up to three levels of tiling. Tiling is required for improving cache and memory access locality and implementing the Hierarchical Z buffer and Z compression algorithms. The higher level can be set to fit a memory page or the size of the framebuffer caches. The second level is used as the scan step by the fragment generator algorithm and the third level is set to the size of the HZ blocks and framebuffer cache lines. In the current implementation the second and third tile levels are both set to 8x8 fragments. We implemented two different Fragment Generators: a fragment scanner that traverses the triangles tile by tile as described for Neon [16] and the recursive rasterization algorithm described by McCool [15]. We currently use the latter as our default fragment generator.

The generated fragment tiles are tested against a Hierarchical Z buffer [17] to remove non visible fragment quads from the pipeline at a very fast rate (up to two 8x8 fragment tiles per cycle in the baseline configuration). The HZ buffer, a single HZ level, is stored as on chip memory to save bandwidth. The buffer requires 256 KB for resolutions up to 4096x4096 with 8 bits of Z precision. The Z reference values for the HZ buffer are calculated when lines are evicted from the Z cache and compressed. Fragments marked as culled by

the fragment generator and outside the scissor window are removed at this stage.

Cache	Size (KB)	Associativity	Lines	Line Size (bytes)	Ports
Texture	**16**	**4**	**16**	**256**	**4x4**
Z	**16**	**4**	**16**	**256**	**4**
Color	**16**	**4**	**16**	**256**	**4**

Table 2: Baseline ATTILA architecture caches

After HZ the generated tiles are divided into smaller 2x2 fragment tiles, named quads, the basic work unit for our fragment processing stages: Z and Stencil Test, Interpolator, Fragment Shader and Color Write. The fragment quad is the work unit in most current GPUs. Working on fragment quads improves the memory access locality and allows an easy implementation of the per fragment derivatives required for the texture mipmap level of detail (lod) computation.

The Z and Stencil unit (ROPz) tests the received fragment quads against the stencil and a depth buffer which stores 8 bits for stencil and 24 bits for depth per element. Quads with all the fragments marked as culled are removed from the pipeline at different points (after Z and stencil test and after shading) while partial quads continue to flow down the pipeline until they are fully culled or update the framebuffer. A Z cache is implemented to exploit access locality to the depth and stencil buffer. The Z cache configuration for the baseline architecture can be found in Table 2. The Z cache implements a lossless compression algorithm with 1:2 and 1:4 ratios to reduce bandwidth usage. Fast Z and Stencil clear, performed in a few cycles and without accessing memory, is also implemented. The algorithms implemented are based on an ATI Hot3D presentation [18] and a related patent [19].

The Interpolator unit interpolates the fragment attributes from the triangle vertex attributes received from Primitive Assembly. We implement the perspective corrected linear interpolation algorithm described in [5] and further specified in the OpenGL API. Some GPUs may interpolate the fragment attributes using the hardware available in the Fragment Shader [4]. The interpolated fragment quads are queued and issued to a Fragment (or Unified) Shader unit.

The Texture Unit attached to each Fragment (or Unified) Shader processes texture requests for a whole fragment quad. A small Texture Cache exploits the high data locality of mipmapping and bilinear filtering to reduce bandwidth usage. The implemented throughput is one bilinear sample per cycle and one trilinear sample every two cycles. We support cube-map and one, two and three dimensional textures. Our Texture Cache architecture is based on the analysis of texture cache architectures from Hakura and Gupta [20], work on texture cache prefetching [21] and proposed implementations [22]. The Texture Cache configuration can be found in Table 2. Relatively small texture caches are known [20] to work well. We support texture compression [24] to further reduce bandwidth usage with our current implementation decompressing the texture blocks into the texture cache.

We decided to remove the alpha test and per fragment fog hardware units from the pipeline and instead implement them as fragment programs. Our OpenGL library creates or modifies the shaders programs as required.

Shaded fragment quads are stored and sent to the Color Write unit (ROPc) where the framebuffer is updated. We implement all the update functions defined in the OpenGL API. The architecture of the Color Write unit is very similar to that of the Z and Stencil test unit with the Color Cache supporting fast color clear of the whole color buffer. The Color cache configuration can be found in Table 2.

The Memory Controller is the unit that interfaces with GPU memory and system memory (AGP or PCI Express). The access to ATTILA memory is based on of the GDDR3 specification. The memory access unit is a 64 byte transaction (4 cycle transfer from a double rate 64 bit DDR channel). The four channels of our baseline architecture provide up to 64 bytes per cycle to the pipeline. The memory modules for each channel are interleaved on a 256 byte basis. Configurable cycle penalties for opening a new memory page, read to write transitions and write to read transitions are implemented. The

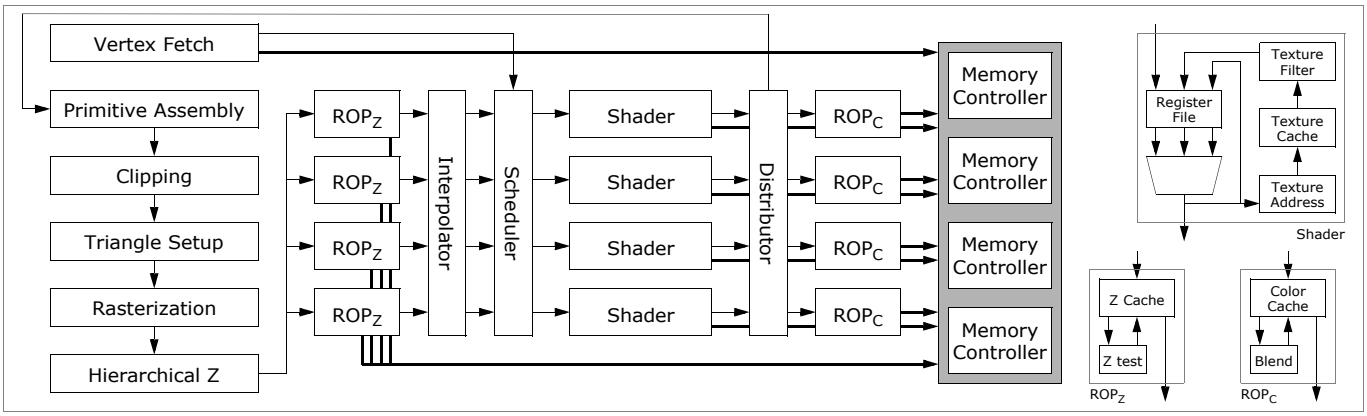

Figure 2. ATTILA Unified Architecture.

system memory bus resembles the PCIe x16 bus (two channels, one for reading data another for writing data). A number of queues and dedicated buses of configurable width conform a complex crossbar that services the memory requests for the different GPU units.

The last hardware unit we implement is the DAC. Although the DAC screen refresh function consumes a relatively small amount of memory bandwidth, we nevertheless wanted to support it in our simulator. However the main task of our DAC unit is to dump the color buffer into a file to allow verification of the rendered output of our architecture against the rendered output by a real GPU.

Our architecture is scaled configuring the number of shader units and fragment quad processing units and their capabilities. The other units are configured so they don't become the bottleneck. Our published work shows examples of how our architecture and simulator can be scaled. In [1] we tested different GPU configurations (unified and non-unified architectures) from a low end GPU to a high end future GPU (current GPUs implement at most 4 or 6 quad fragment shading units). In [2] we inverted the scaling direction and downgraded the simulated architecture to the most basic embedded GPU, configured with a single fragment shader unit doing all the vertex, fragment and triangle shading work.

2.3 Unified Shader

Our unified shader architecture is based on the ISA described at the ARB vertex and fragment program OpenGL extensions [30][31]. The shader works on 4 component 32 bit float point registers and implements SIMD and scalar instructions. The fragment and unified shader shaders implement texture instructions for accessing memory and a kill instruction for culling the fragment. The ARB ISA defines four register banks: input attributes (read only), output attributes (write only), temporal registers (read/write) a constants (read only). Figure 3 shows the shader programming environment.

Shader instructions are stored in a relatively small shader instruction memory that is preloaded before starting the batch rendering. The shader processor pipeline has a fetch stage, a decode stage, an instruction dependant number of execution stages (configurable, currently ranging from 1 to 9 cycles) and a write back stage. The shader processor executes instructions in order and stalls when data dependences are detected.

Our shader architecture implements multithreading to hide instruction execution and texture access latency exploiting the inherent parallelism of the shader inputs, as all shader inputs are completely independent. For the vertex shader target only a few threads are required to hide the latency between dependant instructions, so for our baseline implementation only 12 threads are supported. Fragment and unified shaders require more threads to hide the latency of the texture accesses so we configure up to 112 shader inputs on execution in the baseline architecture. A texture access blocks the thread until the texture operation finishes.

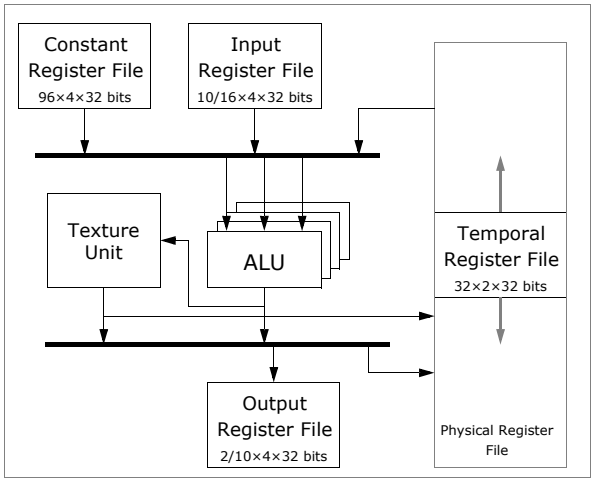

Figure 3. Unified shader architecture.

The availability of threads is further limited by the number of temporal registers that the running shader program uses. The ARB ISA defines up to 32 registers temporal registers but less are required, 4 to 8 for vertex programs and 2 to 4 for fragment programs. We provide a pool of 96 physical registers for non-unified vertex shaders and 448 physical registers for fragment and unified shaders.

Another characteristic of our shader model is that we support, for the fragment and unified shaders, to work on groups of shader inputs as a single processing unit (or single thread). The same instructions are fetched, decoded and executed for a group of four inputs. The current implementation for texture accesses requires groups of four fragments to be processed in parallel as described in section 2.2. The four inputs that form a group are processed in parallel and the shader unit works as a 512 bit processor (4 inputs, 4 components, 32 bit per component). Our fetch stage can be configured to issue one or more SIMD and scalar instructions for an input group per cycle. For the non-unified vertex shader each input is a thread as they are reported to work in current GPU [3].

3. ATTILA Simulator

We have developed a highly accurate, cycle-level and execution driven simulator for the GPU architecture described in the previous section. The model is highly configurable (the configuration files for our architecture has over 100 parameters) and modular, to enable fast yet accurate exploration of microarchitectural alternatives. From a software engineering point of view, the model is structured on top of two fundamental abstractions: boxes and signals, following [28]. Boxes are a module that abstracts a "large enough" piece of the pipeline, for example the Clipper or the Fragment Generator. Signals are the "wires" that connect the different boxes in the pipeline. All communication between boxes happens in a message passing style by sending information through a signal wire.

Because signals have an associated latency (in clock cycles) and bandwidth correctly modelling, and checking, communication delays and pipeline stages is very straightforward, yet still highly configurable.

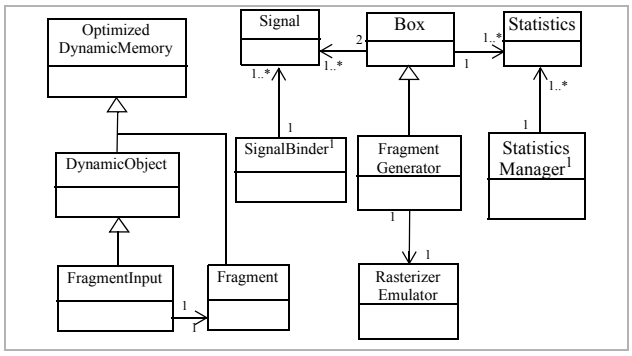

Figure 4. Simulator Classes Diagram

The simulator is written in C++ implementing the box, signal and associated simulation concepts as a framework of C++ classes (Figure 4). All simulated boxes derive from the generic Box class. The SignalBinder static class is used as a name server for registering and associating, using unique names, signals with the boxes they connect. All the statistics collected by the simulator derive from a Statistics template class. The StatisticManager static class serves as a name server to register, update, gather and output those statistics. The DynamicObject class is used to track the objects that travel through the signals. The OptimizedMemory class, from which the DynamicObject class derives, implements cheap object creation and destruction. All objects that travel through signals derive from the Dynamic Object class. The DynamicObject class stores information (an identifier, a 'color' and a text string) about the objects. The identifier is used to associate related objects, forming a multilevel hierarchy. For example fragments are associated with a triangle so a memory access associated with the fragment would also be associated with the triangle. This information can be dumped each cycle, at the signal exit, as a signal trace file used with the Signal Trace Visualizer tool to debug the simulator performance.

The model specifies that a box can use local data, accessible in one cycle from input signals or registers and queues, to update the box state and send outputs to the next box. Signals feed data to the attached box limited by the configured latency, cycles, and bandwidth (in objects) and perform verification checks that may terminate the simulator, for example when bandwidth is exceeded or data is lost. Boxes and signals are updated every cycle. Boxes simulate the architecture resource restrictions, control and data flow while signals simulate latency and bandwidth. To reduce the number of required boxes we usually implement in the same box related tasks that can be performed in parallel, for example receiving a new input while sending a new output from separated queues. Signals are also used to simulate the latency of multistaged pipelines that do not require a more precise simulation. Multistage ALUs are simulated in this manner. The box where the computation starts decides the latency of the computation, if the execution latency is variable, and simulates any limitation in the ALU input or output througput.

The signals registered by a box in the SignalBinder conform the box interface. Signals are registered with a unique name, a direction (input or output), a bandwidth and a latency. The model allows to easily replace a box with another box implementing an alternative architecture registering the same

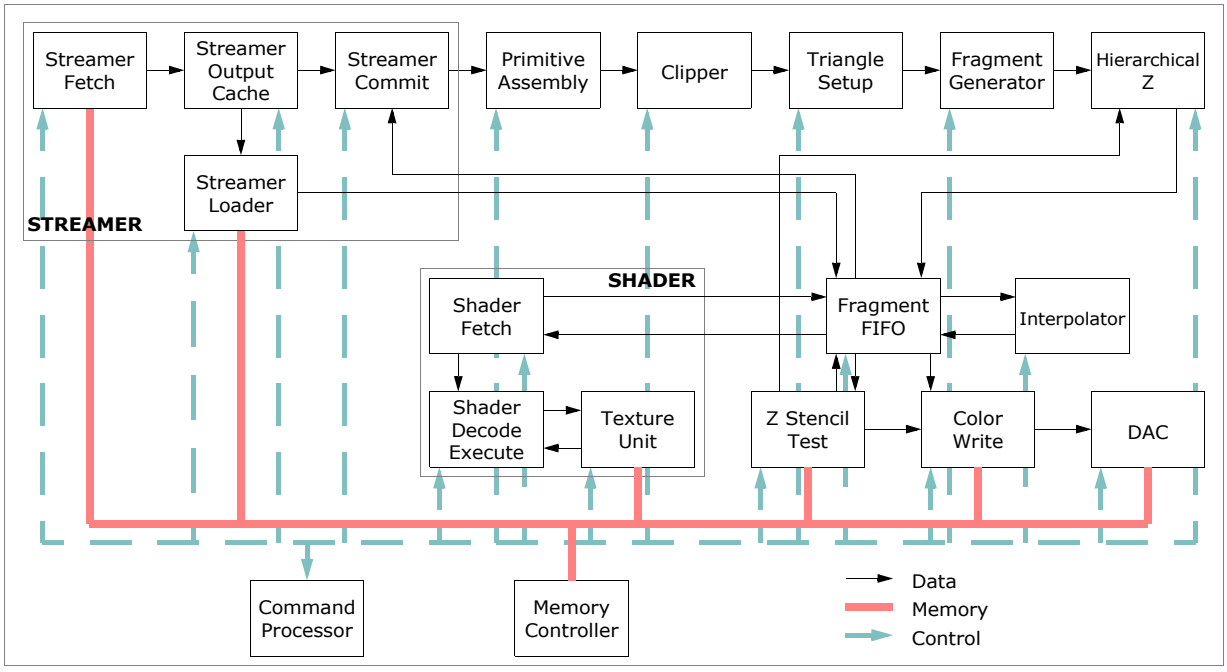

Figure 5. Diagram of ATTILA (unified shader model) simulator boxes and signals.

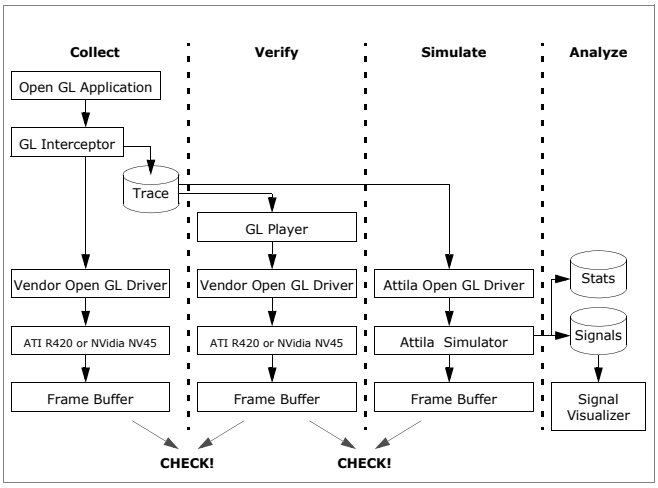

Figure 6. ATTILA Simulation Framework.

signals and supporting the same input and output objects.

Figure 5 shows the current mapping of the ATTILA pipeline implementing the unified shaders model to the boxes and signals model. Not all signals are shown in the figure but the missing signals are mostly feedback signals and intra box signals that simulate multistage ALUs. The granularity at which the GPU pipeline stages are mapped into boxes varies. For example four boxes were implemented for the Streamer stage while only one box was implemented for Z and Stencil Test and Color Write. As we were experimenting with the model we tried different granularities (the Streamer being the first pipeline stage implemented) and we finally made the decission of packing all closely related behavior and tasks as single boxes (for example Color Write simulates all the process related with of updating the color buffer).

Our caches are currently implemented using a non-signal (method based) interface and attached to a parent box (not show, but for example the Texture Cache box is attached to the Texture Unit box). This allows to simulate single cycle tag access and single cycle data access caches as implementable for the relatively low clock frequencies and small cache sizes of current GPU. The caches are still implemented with the box class and we plan to implement, in the future, a signal based interface to simulate more aggressively clocked GPUs.

Comparing Figure 5 and Figure 2 we can see how the boxes relate with the different pipeline stages. The Fragment FIFO box (a legacy name) corresponds to a crossbar and scheduler that receives input vertices and fragments from producing boxes (Streamer Loader for vertices, Hierarchical Z test, Interpolator and Z Stencil Test for fragments), feeds those inputs into the unified shader boxes, receives the shaded outputs from the unified shader boxes and sends the outputs to the consuming boxes (Streamer Commit for vertices, Z Stencil Test or Color Write for fragments). The FragmentFIFO box also implements the two datapaths required to perform the Z and Stencil test before and after fragment shading.

The simulator is “execution driven” in the sense that real data travels through signals from box to box. For example, the actual 32-bit FP attributes for a vertex travel through the timing simulator pipeline. Boxes use data received from signals and data stored on local structures to call an associated functional library that may create new or modify the input data. The output data then flows to the next pipeline stages (boxes). With this functional emulation of the pipeline the simulator is generating the same (or equivalent) accesses to memory, hits and misses and bandwidth usage that a real GPU would. The final result of the functional emulation is used later to verify the correctness of the simulation.

The emulation libraries, implemented as separated C++ classes, implement all the rendering computations. We divided the rendering emulation in the following classes: ShaderEmulator, TextureEmulator, FragmentOperatorEmulator and ClipperEmulator. The ShaderEmulator implements a threaded interpreter that executes, instruction by instruction, shader programs updating the stored per thread state (registers). The Shader Emulator is used by the shader boxes: ShaderFetch and ShadedDecodeExecute. The TextureEmulator calculates memory addresses for texture accesses, calculates the number of samples for anisotropic filtering, converts texel data into the internal format and filters the sampled texel data. It also implements decompression functions for compressed textures. The TextureEmulator is used by the TextureUnit box. The FragmentOperatorEmulator implements the Z and Stencil test functions, the compression algorithms for the Z cache and the Color Write blend and update functions. The Clipper Emulator only implements a basic trivial rejection test for triangles completely outside the frustum volume.

The emulation libraries help to simulate different microarchitectures, but that implement the same functionality, for the same GPU stage as only the timing simulator code, the boxes and signals, must be reimplemented. It also keeps emulation and simulation related bugs separated. When bugs in the emulator code are removed and the emulator becomes relatively stable bugs are likely to come only from changes in the simulation code. Another benefit of the emulator libraries is the possibility, in the future, of implementing a light emulator to skip fast through regions of graphic traces while performing the rendering and either a basic pipeline simulation, trace profiling or pre setting the simulator (filling caches and memory).

4. OpenGL Framework

A goal for the ATTILA framework was to run real graphic applications on our detailed timing simulator. To this end, we developed an OpenGL framework for our ATTILA architecture (D3D is in the works). We have implemented an important part of the OpenGL API with our own C++ object oriented library and an autogenerated library of cover functions. Figure 6 shows the framework and the process of collecting traces from real graphic applications, verifying and

simulating the trace and verifying the simulation result.

The ATTILA Command Processor supports a simple set of instructions: write a render state register, write a buffer into GPU memory, draw a batch, fast clear of the color or z and stencil buffers and swap the current front and back color buffers (finishing the frame). Our OpenGL framework bridges the gap between the OpenGL API and the ATTILA architecture translating each OpenGL API call into one or more of these low-level control commands. The framework software organization is layered: the top layer, the library, manages the OpenGL state while the lower layer, the driver, offers basic services as writing registers, sending commands, configuring shaders and basic memory allocation.

The features supported by our OpenGL framework include: basic OpenGL functionality (about 200 API calls supported); ARB Vertex and Fragment program extensions; vertex arrays and buffer objects; full legacy vertex and fragment fixed function API and emulated alpha test and fog support using driver generated shader programs (partly based on [27]); full support for texturing (texture targets, mipmapping, filtering modes, wrap modes multitexture, texture objects); and per fragment operations (stencil test, Z test and blending functions).

The ATTILA OpenGL library implements a memory abstraction component for the different OpenGL objects: programs, textures and vertex buffers. MemoryObject offers high-level methods for allocating to, synchronizing with and deallocating from the ATTILA memory, freeing the library programmer from the challenging job of implementing and verifying a memory replacement policy for every object type.

GLInterceptor, see Figure 6, replaces the OpenGL library and records all OpenGL commands issued by the application with all their parameter values, associated texture and vertex buffers data. This information is stored in an output file, a trace file for our simulator. All the OpenGL commands and data are also passed to the original library to continue the application execution. To verify the integrity and faithfulness of the recorded trace a second tool, GLPlayer, can be used to reproduce and validate the captured trace. In the current implementation the trace isn't time stamped so our simulator is isolated from non GPU system related effects (CPU limited executions, disk accesses, memory swapping). Our simulator uses the simulated DAC unit, as in a real GPU to the CRT or LCD, to dump the rendered frame into a file. We use the dumped frame for verifying the correctness of our simulation and functional emulation.

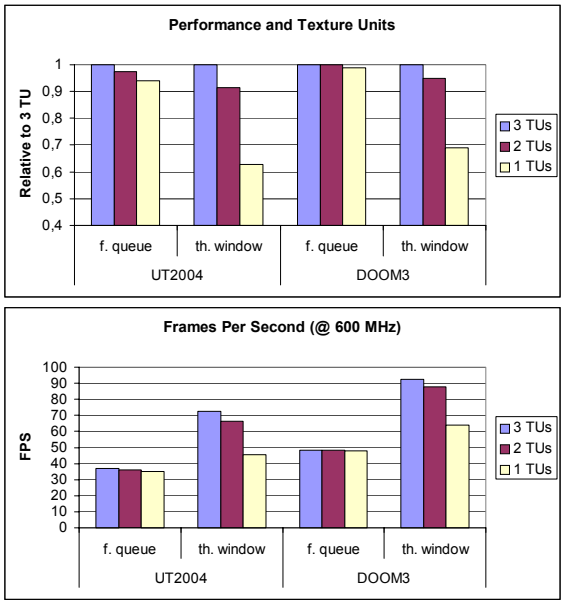

Figure 7. Shader ALUs vs. Texture Units

The ATTILA simulator generates two more outputs: a statistics file in CSV format storing detailed information about all events occurring within our ATTILA microarchitecture, including low level resource utilization of all the pipeline stages described in section 2, cache hit and miss ratios, memory bandwidth usage, etc.; and a signal trace file used by Signal Trace Visualizer tool for debugging the simulated microarchitecture. We currently support up to 300 different statistics from all the implemented pipeline stages.

The last feature of our OpenGL framework is the implementation of a "hot start" technique to start the simulation at any frame of the trace file. As frames are independent from each others groups of frames can be simulated independently. The driver skips over the draw commands and only sends state changes and buffer writes to the simulator. The is useful for simulating traces of billions of cycles and hundreds of frames using a PC cluster with hundreds of nodes.

5. A Simple Case Study: Shader ALUs VS Texture Units

We have selected a simple case test to show some of the features in the current implementation of the simulator. We want to determine how performance degrades when we change the relatively constant 1:1 ratio between fragment shader ALUs (ignoring multi issue ALU setups) and texturing capabilities in current GPUs. ATI has recently presented the RV530 GPU implementing a 3:1 ALU to texture ratio.

The test base configuration implements three unified shaders, one ROP and two 64-bit DDR buses to GPU memory. We test two different configurations: a thread window (1 thread = 1 fragment quad or 4 vertices) enabling out-of-order execution of the shader threads and a shader input queue that only allows in-order execution of the shader inputs. The thread window and the shader input queue are global to the three shader units and both store a maximum of 384 inputs (96 threads or 96 quad inputs). The associated register bank stores up to 1536 temporal registers. For both configurations we evaluate the performance when the number of Texture Units

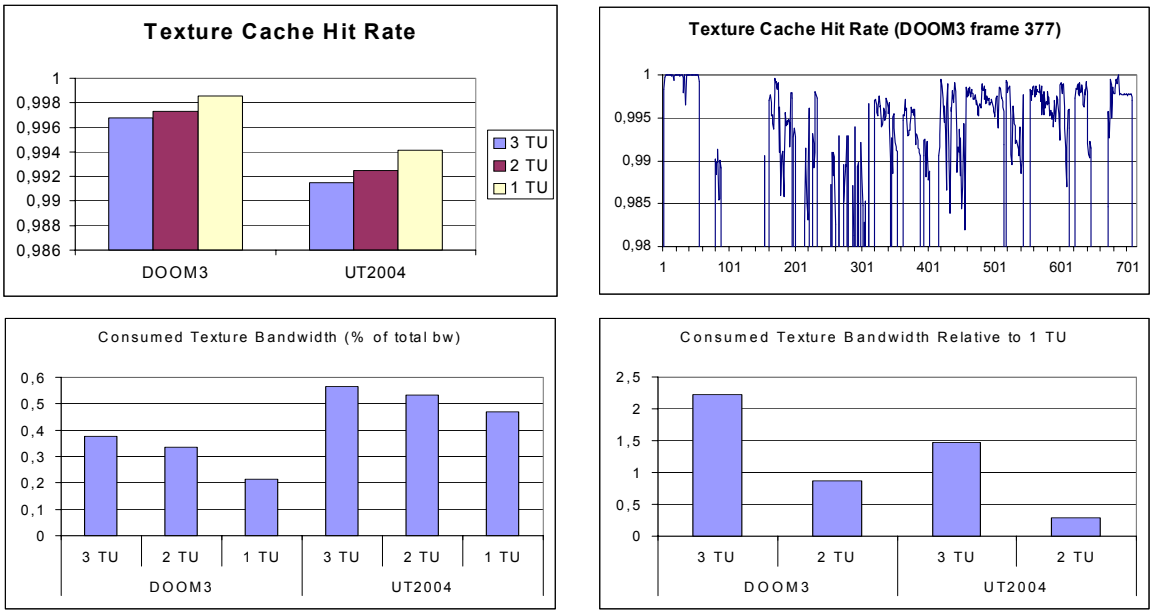

Figure 8. Texture Cache Hit Rate and Consumed Texture Bandwidth

varies from 3 to 1. The graphic traces selected to for the test are 40 frames from a UT-2004 Primeval timedemo trace and 40 frames from a DOOM3 trDemo2 timedemo trace. Both traces are rendered at 1024x768 and with Anisotropic Filtering set to a maximum of 8 samples. The selected frames correspond with around 300-700 million simulation cycles and a simulation time of 20 hours on a P4 Xeon at 2 GHz.

Figure 7 shows the performance degradation and frame rate (frames per second for GPU and memory clocked at 600 MHz) when configuring from 3 to 1 TUs. In the thread window configuration performance takes a small hit (5-10%) for the 3 TUs to 2 TUs transition, and a relatively large hit from 3 TUs to 1 TU. For the shader input queue configuration the queue size is too small to hide the latency of texture memory accesses and the number of TUs doesn't affect performance.

In Figure 8 we display different statistics that show how the texture cache hit rate and the texture bandwidth changes for the thread window configuration. Fragment quads assigned to each texture unit may come from overlapping regions and the same texture data is requested by multiple texture units. The work distribution algorithm currently implemented in the simulator, not properly optimized, has also an effect in increasing the number of misses to the texture cache and the consumed memory bandwidth. The texture cache hit rate for the 3 TUs configuration, sampled each 10K cycles, for a DOOM3 frame is also shown.

Figure 9 shows the workload characterization, sampled each 10K cycles, for a DOOM3 frame for the configurations (top to bottom): thread window with 3 TUs, thread window with 1 TU, and the shader input queue with 3 TUs. For the shader input queue all the GPU units are under utilized because of not being able to hide the texture memory access latency. When comparing the two thread window configurations we can clearly see that with 1 TU the GPU is completely limited by texturing, with a 95-99% utilization rate of the TU.

Figure 10 shows frame 802 from the DOOM3 trDemo2 trace and compares the differences between a real GPU, NVidia GeForce 5900, and our OpenGL Framework and ATTILA simulator. Comparing those images three rendering bugs (at the time they were captured) in the ATTILA simulator can be found: a bug in DXT3 alpha channel decompression, a bug clamping negative fragment shader output colors and a bug in stencil clear implementation.

6. Related Work

NVidia presented the implementation of a vertex shader for the GeForce3 (NV2x) GPU [3]. Information about shader architecture can be obtained from patents [4] complemented with the analysis of the performance of the shader units in current GPUs [35]. More actualized information about NVidia and ATI implementations surfaces as unofficial or unconfirmed information on Internet forums [5][6]. T. Aila et al. proposed delay streams [7] to improve the performance of immediate rendering GPU architectures with minor pipeline modifications. Akenine-Möller described a graphic rasterizer for mobile phones [8]. Numerous research papers evaluate GPUs as general purpose stream processors [32][33][34] and for implementing raytracing [12].

OpenGL wrappers, as Chromium [13], or open source implementations, MESA [37], can be used to profile graphic applications and perform simulation of the graphic pipeline. GLSim [9] and GLTrace tools from Stanford are basic but relatively outdated simulation tools. QSilver [10][36], based on Chromium, can simulate traces from whole games. QSilver

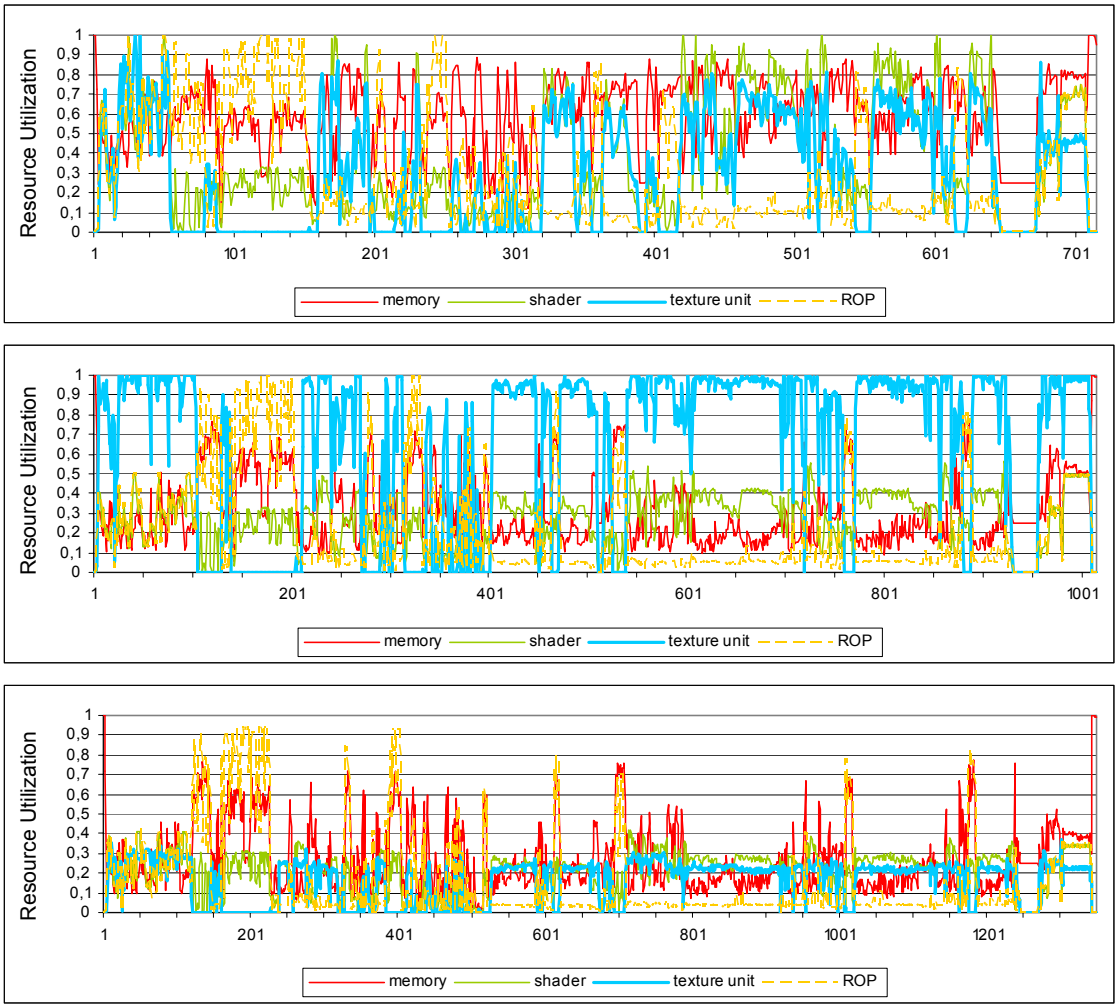

Figure 9. Workload characterization (resource utilization) for DOOM3 frame 377. From top to bottom: thread window with 3 TUs, thread window with 1 TU and shader input queue with 3 TUs.

feeds a trace, captured with Chromium, that has been preprocessed and annotated with additional information (information obtained from hardware counters in a real GPU, for example pixels drawn) to a cycle-timer simulation model. The current QSilver implementation is used to study the power and thermal behavior of current GPUs. The simulated model is relatively simple and based on statistics and probability distributions but could in the future (as it's open source) be expanded to support more complex simulation models.

7. 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].

8. Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIC2001-0995-C02-01 and TIN2004-07739-C02-01 and by the Department of Universities, Research and Society of the Generalitat de Catalunya and the European Social Fund.

Figure 10. Frame 802 from DOOM3 trDemo2. Left from NVidia GeForce 5900, right from ATTILA simulator.

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/