Shader Performance Analysis on a Modern GPU Architecture

Victor Moya[1], Carlos Gonzalez, Jordi Roca, Agustin Fernandez, Roger Espasa[2]
Department of Computer Architecture, Universitat Politècnica de Catalunya
{vmoya, cgonzale, jroca, agustin, roger}@ac.upc.edu

Abstract

This paper presents an analysis of the performance of the shader processing units in a modern Graphics Processor Unit (GPU) architecture using real graphic applications. The architecture of a modern GPU is described and a simulator and associated framework used to evaluate the architecture is introduced. The paper analyses the effects in performance of different configurations of the shader processing units and compares a classic GPU with a unified shader GPU. The evaluated unified shader architecture proves to be 15% to 30% more efficient, in terms of area, with a 2% to 7% improvement in performance when compared with a similar non-unified architecture.

1. Introduction

The microarchitecture of the shader units in modern GPUs is becoming a key research topic as they evolve to support more complex and generic programming models. Recently presented by ATI and MS the GPU for the XBOX360 game platform implements a unified shader architecture. However, there are no proper evaluations of the performance differences between the classic GPU architecture (implementing separated vertex and fragment shader units) and a unified shader GPU architecture. The present work compares the performance and efficiency of both architectural models along with other parameters of the architecture.

We have developed a generic GPU microarchitecture that contains most of the advanced hardware features seen in today's major GPUs. We have liberally blended techniques from all major vendors and also from the research literature [24], 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 also produced an OpenGL framework (library, driver and trace capture tool) able to run full applications (i.e., commercial games) on our GPU microarchitecture. Finally we have used this simulator to evaluate the performance of a unified shader architecture and basic performance parameters of the shader microarchitecture.

The reminder of this paper is organized as follows: Section 2 introduces the 3D rendering algorithm. Section 3 describes the GPU pipeline and briefly discusses our ATTILA GPU architecture. Section 4 introduces the simulator and the associated OpenGL framework. Section 5 describes in detail the architecture of the shader units and the unified shader programming model. In Section 6 the performance and efficiency of different shader architecture configurations is evaluated using a UT2004 trace. Finally Sections 7 and 8 present related work and conclusions.

2. 3D Rendering

GPUs are designed as specific purpose processors implementing a specific 3D rendering algorithm. The 3D rendering algorithm implemented takes as input a stream of vertices that defines the geometry of the scene. The input vertex stream passes through a computation stage that transforms and computes some of the vertex attributes generating a stream of transformed vertices. The stream of transformed vertices is assembled into a stream of triangles, each triangle keeping the attributes of its three vertices. The stream of triangles may pass through a stage that performs a clipping test. Then each triangle passes through a rasterizer that generates a stream of fragments, discrete portions of the tri-

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.
[1] Research work supported by the Department of Universities, Research and Society of the Generalitat de Catalunya and the European Social Fund.
[2] Intel Labs Barcelona (BSSAD), Intel Corp.

angle surface that correspond with the pixels of the rendered image. Fragment attributes are derived from the triangle vertex attributes.

This stream of fragments may pass through a number of stages performing a number of visibility tests (stencil, depth, alpha and scissor) that will remove non visible fragments and then the stream of fragments will pass through a second computation stage. This second fragment computation stage may modify the fragment attributes using additional information from n-dimensional arrays stored in memory (textures). The stream of shaded fragments will, finally, update the framebuffer.

Modern GPUs implement the two described computation stages as programmable stages named vertex shading and fragment shading. The programmability of these stages and the streaming nature of the rendering algorithm allows the implementation of other stream based algorithms over modern GPUs [30][31]. However those implementations may not be optimal. The non programmable stages are configurable using a limited and predefined set of parameters.

The shading stages are programmed using a shader, or shader program, a relatively small program written in either assembly-like (legacy) or high level C-like languages for graphics that describes how the input attributes of a processing element (a vertex or a fragment) are used to compute its output attributes.

Graphics applications use software APIs (OpenGL or Direct3D) that present an interface for the described rendering algorithm and map the algorithm to the modern GPU hardware capabilities.

The 3D rendering algorithm is embarrassingly parallel and shows parallelism at multiple levels. The largest source of parallelism comes from the data and control independency of the processing elements: vertices are independent of each other, triangles are mostly independent (except for transparent surfaces) and fragments from the same triangle are independent.

GPUs exploit four forms of parallelism: the pipeline is divided into hundreds of single cycle stages to increase the throughput and the GPU clock frequency (pipeline parallelism); the pipeline stages are replicated to process in parallel multiple vertices, triangles and fragments (data parallelism); multiple processing elements are stored and processed concurrently to hide memory latencies in a specific stage or processing unit (multithreading); and independent instructions in a shader program may be executed in parallel (instruction level parallelism).

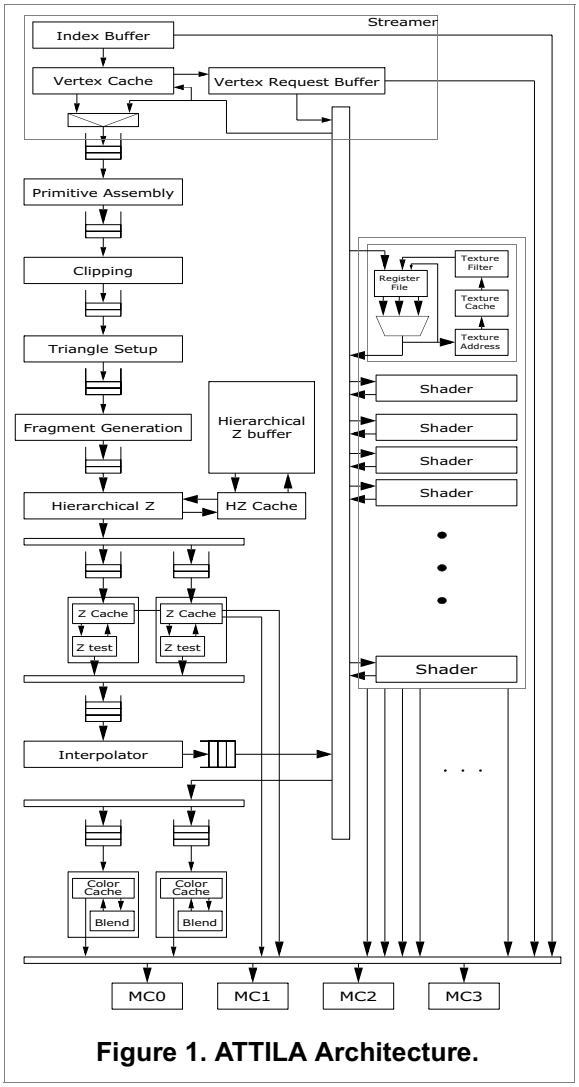


Figure 1. ATTILA Architecture.

3. ATTILA Architecture

We will now briefly describe our ATTILA implementation of the 3D rendering pipeline. We have blended techniques and ideas from different vendors and publications [24] and we have made educated guesses in those areas where information was specially scarce. Our implementation correlates in most aspects with current real GPUs.

The ATTILA architecture supports both hard partitioning of vertex and fragment shaders (the norm in current GPUs) or a unified shader model (that will be implemented in future GPUs). Figure 1 shows the ATTILA GPU graphic pipeline for the unified shader model. The input and output processing elements, the bandwidth and the latency in cycles of the different ATTILA stages can be found at Table 1. Table 2 shows

the sizes of some of the input queues in those stages and the number of threads supported in the vertex and fragment/unified shader units. The diagram and the table data corresponds to a reference architecture implementing 4 vertex shaders (non unified), 2 shader units (fragment or unified), 2 ROPs and four 32-bit DDR channels.

Two GPU units are not shown in Figure 1 the Command Processor that controls the whole pipeline, processing the commands received from the system main processor and the DAC unit that consumes bandwidth for screen refreshes and outputs the rendered frames into a file.

Table 1: Queue sizes and number of threads in the ATTILA reference architecture

Unit	Size	Element width
Streamer	48	16×4×32 bits
Primitive Assembly	8	3×16×4×32 bits
Clipping	4	3×4×32 bits
Triangle Setup	12	3×4×32 bits
Fragment Generation	16	3×4×32 bits
Hierarchical Z	64	(2×16+4×32)×4 bits
Z Test	64	(2×16+4×32)×4 bits
Interpolator	-	-
Color Write	64	(2×16+4×32)×4 bits
Shader (vertex)	12+4	16×4×32 bits
Shader (fragment/unified)	112+16	10×4×32 bits

The Streamer unit reads streams of vertex input attributes from GPU or system memory and feeds them to a pool of vertex or unified shader units (Figure 1). The streamer also supports an indexed mode that allows reusing vertices shaded and stored in a small post shading cache. After shading the Primitive Assembly stage converts the shaded vertices into triangles and the Clipper stage performs a trivial triangle rejection test.

The rasterizer stages generate fragments from the input triangles. The rasterization algorithm is based on the 2D Homogeneous rasterization algorithm [14] which allows for unclipped triangles to be rasterized. The Triangle Setup stage calculates the triangle edge equations and a depth interpolation equation while the Fragment Generator stage traverses the whole triangle generating tiles of fragments. ATTILA supports two fragment generation algorithms: a tile based fragment scanner [16] and a recursive algorithm [15] (used for the paper experiments).

After fragment generation a Hierarchical Z buffer [17] is used to remove non visible fragment tiles at a fast rate without accessing GPU memory. The HZ buffer is stored as on chip memory and supports resolutions up to 4096x4096 (256 KB).

The processing element for the next stages is the fragment quad, a tile of 2x2 fragments. Most modern GPUs use this working unit for memory locality and the computation of the texture LOD in the Texture Unit.

The Z and stencil test stage removes as early as possible non visible fragments thereby reducing the computational load in the fragment shaders. Figure 1 shows the datapath for early fragment rejection. However another path exists to perform the tests after fragment shading. ATTILA only supports a depth and stencil buffer mode: 8 bits for stencil and 24 bits buffer for depth. The Z and Stencil test unit implements a 16 KB, 64 lines, 4-way set associative cache. The cache supports fast depth/stencil buffer clear and depth compression. The architecture is derived from the methods described for ATI GPUs [18][19].

The Interpolator unit uses perspective corrected linear interpolation [5] to generate the fragment attributes from the triangle attributes. However other implementations may interpolate the fragment attributes in the Fragment Shader [4]. The interpolated fragment quads are fed into the fragment or unified shader pool. The Texture Unit attached to each fragment or unified shader supports n-dimensional and cubemap textures, mipmapping, bilinear, trilinear and anisotropic filtering. The Texture Cache architecture is based on [20][21][22] and is configured as a 64 lines, 4-way set associative, 16 KB cache. Relatively small texture caches are known to work well [20]. Compressed textures are also supported [23].

Table 2: Inputs, outputs and latencies in cycles in the reference ATTILA architecture

Unit	Input BW	Output BW	Latency
Streamer	1 index	1 vertex	Mem
Primitive Assembly	1 vertex	1 triang.	1
Clipping	1 triang.	1 triang	6
Triangle Setup	1 triang.	1 triang	10
Fragment Generation	1 triang	2×64 frag.	1
Hierarchical Z	2×64 frag.	2×64 frag.	1
Z Test	4 frag.	4 frag.	2+Mem
Interpolator	2×4 frag.	2×4 frag.	2 to 8
Color Write	4 frag.		2+Mem
Shader (vertex)	1 vertex	1 vertex	variable
Shader (fragment/unified)	4 frag.	4 frag.	variable

The Color Write stage basic architecture is similar

to the Z and Stencil test stage architecture but color compression is not supported.

The Memory Controller interfaces with the ATTILA memory and the main computer memory system. The ATTILA memory interface simulates a simplified GDDR memory where banks are not being modeled. The memory access unit is a 64 byte transaction: a single 4 cycle 8 32-bit word burst from a single GDDR channel. The number of channels and the channel interleaving is configurable. Read to write and write to read penalties are implemented. A number of queues and dedicated buses conform a complex crossbar that services the memory requests for the different GPU stages.

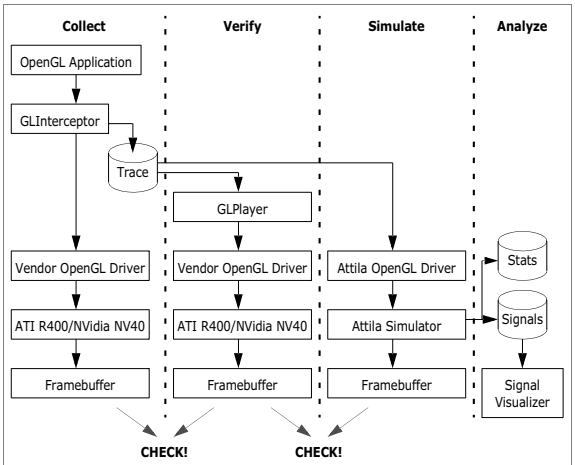


Figure 2. ATTILA Simulation Framework.

4. ATTILA Simulator and OpenGL Framework

We have developed a highly accurate, cycle-level and execution driven simulator for the ATTILA architecture described in the previous section.

The model is highly configurable (over 100 parameters) and modular, to enable fast yet accurate exploration of microarchitectural alternatives.

The simulator is "execution driven" in the sense that real data travels through signals from box to box. A box uses the data received from signals and the data it stores on its local structures to call the associated functional module that creates new or modified data that continues flowing through the pipeline. The same (or equivalent) accesses to memory, hits and misses and bandwidth usage that a real GPU are generated. This key feature of our simulator allows to verify that the architecture is performing the expected tasks.

Our simulator implements a "hot start" technique that allows the simulation to be started at any frame of a trace file. Frames, disregarding data preload in memory, are mostly independent from each other and groups of frames can be simulated independently. A PC cluster with 80 nodes is used to simulate dozens of frames in parallel. The current implementation of the simulator can simulate up to 50 frames at 1024x768 of a UT2004 trace, equivalent to 200-300 million cycles, in 24 hours in a single node (P4 Xeon @ 2 GHz).

We have developed an OpenGL framework (trace capturer, library and driver) for our ATTILA architecture (D3D is in the works).

Figure 2 shows the whole framework and the process of collecting traces from real graphic applications, verifying the trace, simulating the trace and verifying the simulation result.

Our OpenGL stack bridges the gap between the OpenGL API and the ATTILA architecture translating each OpenGL call into one or more low-level control commands and maintaining and updating OpenGL state. The driver software organization is layered: the top layer manages all OpenGL state while the lower layer offers basic services to configure the graphics hardware and a basic memory allocation model. The features supported by our OpenGL library are: basic OpenGL functionality, about 200 API calls supported; ARB Vertex and Fragment program extensions; vertex arrays and vertex buffer objects; legacy vertex and fragment fixed function API emulated with library generated shader programs [25]; texturing; stencil test, Z test and blending functions; and alpha test and fragment fog emulated using library generated shaders.

The GLInterceptor tool uses an OpenGL stub library to capture a trace of all the OpenGL API calls and data that the graphic application is generating as shown in Figure 2. All this information is stored in an output file (a trace). To allow the graphic application continue its normal execution GLInterceptor also passes on all the OpenGL commands and data to the original library. To verify the integrity and faithfulness of the recorded trace a second tool, GLPlayer, can be used to reproduce the trace.

After the trace is validated, it is feed by our OpenGL stack into the simulator. Using traces from graphic applications isolates our simulator from any non GPU system related effects (for example CPU limited executions, disk accesses, memory swapping). Our simulator uses the simulated DAC unit to dump the rendered frames into files. The dumped frame is used for verifying the correctness of our simulation and architecture.

5. Unified Shader Architecture

Our shader architecture follows the OpenGL ARB specifications for vertex [27] and fragment [28] shader programs.

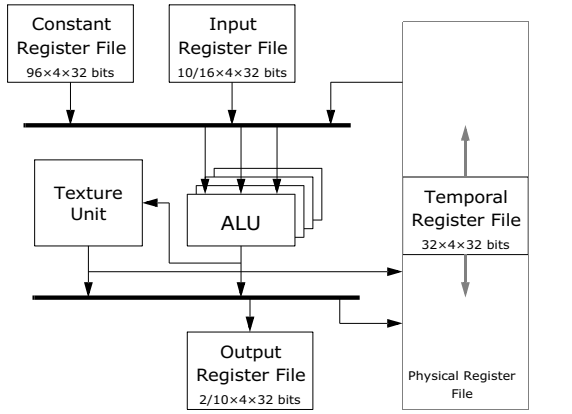


Figure 3. Shader architecture.

The ARB vertex and fragment program specifications define assembly alike instructions that can be used to program how the vertex and fragment output registers can be calculated from per vertex and fragment input registers and a set of per batch constant parameters. There are four defined register banks (as shown in Figure 3): the input register bank, a read only bank, stores the vertex and fragment input attributes; the output register bank, write only, stores the vertex and fragment output attributes; the temporal register bank, which supports reading and writing, is used to store intermediate values; and a constant parameter bank stores parameters that are constant for a whole batch. A shader register is a 4 component 32-bit float point vector, limiting the ARB shader program models to support only float point data. The program size is limited to a few hundred instructions and changes in the execution flow control (loops, branches, functions) are not supported.

The OpenGL specification supports a high level shader language, glSlang [29], that offers additional programming features (vertex textures, looping, branching, functions) but our OpenGL framework doesn't support it yet. The glSlang programming language virtualizes all the hardware resources available for the shader architecture and tasks the compiler and optimizer with accommodating the requested resources with the resources available in the target architecture.

The ARB instructions are defined as an operation opcode, a destination operand and up to three source operands. The source operands support full per component swizzling and negation and absolute value modifiers. The destination operand supports full per component swizzling and masking of the operation result. The ISA supports two types of operations: 4 component SIMD operations, (ADD, CMP, DP3, DP4, MAD, etc.) and scalar operations (COS, EX2, RCP, etc.).

There are a few differences between the vertex and fragment program specifications. Fragments can access texture data with the TEX, TXB, and TXP instructions while vertices can't. Texture instructions, in our architecture, use the SIMD ALU for the texture address computation and then the texture request is issued to the Texture Unit. The Texture Unit processes the request, accesses the Texture Cache and, optionally, memory and performs the filtering of the sampled texels. For fragment programs a KILL instruction is defined, used to 'stop' the processing of a fragment Texture and KILL instructions use vectorial operands. An additional instruction modifier _SAT is defined only for fragment programs to inexpensively implement the required clamping (to the [0, 1] range) of color result values.

Our unified shader architecture implements the super set of both vertex and fragment program models, however our current OpenGL framework is limited to the ARB vertex and fragment program features.

The support for a non unified shader model is implemented capping a unified shader unit to work as a vertex shader unit from a current GPU would. The unified model not only creates a coherent programming model for both fragment and vertex processing but also simplifies the architecture design, and allows a more flexible and efficient use of the shading units. As the workload balance of vertices and fragments changes from batch to batch more shader units can be allocated to the more demanding task. The unification of the vertex and fragment programming models is the target for future APIs (Shader Model 4.0 [26] in Direct3D and OpenGL glSlang) and GPU architectures.

The abundance of parallelism inherent to shader processing (all processing elements are always independent) is exploited via multithreading. The reference architecture used for the paper experiments implements 128 threads per (unified or fragment) shader unit to hide texture (memory) access latency. The vertex shaders for the non unified architecture implement 12 threads to hide the instruction execution latency. The ARB programming model specifies a relatively large number of temporal registers, but in most cases most of those registers aren't used by the shader program. For a more efficient use of the available transistor budget we have implemented per program static allocation of tem-

poral live registers from a per shader unit physical register file to each thread. The number of available threads for multithreaded execution changes as the shader program requirements for live temporal registers changes. The reference architecture implements 4 temporal registers per thread and the allocation granularity is set at two registers per thread.

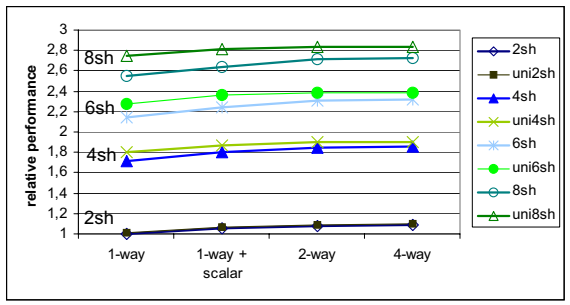


Figure 4. Performance scaling number of shader units and issue width.

The shader units process in parallel groups of four threads (each thread corresponding with a vertex or a fragment) because of a requirement of fragment processing (texture LOD derivative computation). A single PC is kept per group to fetch and issue the same instruction for the four threads implementing an additional SIMD level to the architecture. For the non unified vertex shader architecture no grouping of vertices is performed and each thread has an associated state and PC. A group may be in one of four states: free (no fragments or vertices allocated), ready (instructions can be fetched), blocked or finished (waiting for the thread results to be sent out of the shader unit).

We support two configurable fetch and issue modes in our simulated architecture: fetch and issue of a SIMD instruction and a scalar instruction per cycle and group or fetch and issue of 1 to n instructions (disregarding the type) per cycle and group. Texture instructions can only be issued one per cycle and group and after decode the group becomes blocked, waiting for the Texture Unit to return the result.

The shader instruction decoder detects dependencies and conflicts accessing the register bank ports and may request the shader fetch unit to refetch instructions that can't be issued in a given cycle. Instructions are fetched and issued for any group that is ready in a per shader unit thread group window, supporting the disordered execution of groups. Instructions in the execution flow of a group are always fetched and issued in order. The instructions are fetched from a small sized (not below 512 instructions) instruction memory where shader programs are explicitly loaded before starting the rendering batch.

The shader execution pipeline consists of the following single cycle stages: a fetch stage; a decode stage; a register read stage; a variable number of execution stages (1 to 9 depending on the instruction latency) and a register write back stage. Split hardware pipelines are implemented to receive the shader inputs (vertex and fragment input attributes) from the feeding stage (streamer or interpolator) and send the shader results (vertex and fragment output attributes) to the next rendering stages (post shading vertex cache or the ROP units).

GPU hardware vendors don't disclose the number of supported threads or temporal registers in their architectures. However they do disclose information about the organization of the shader hardware ALUs. While there are many differences between the GPU architectures, most allow grouping multiple ARB instructions (up to 5 or 6) in single cycle issue to a number of parallel SIMD ALUs (even implementing partitioned 2+2 operations), scalar ALUs and special 'mini-ALUs' implementing per operand modifier operations (e.g. vector normalization). Our OpenGL framework doesn't support such level of optimization and the implemented arrangement of ALUs in the shader units is relatively simple (swizzle, modifier, SIMD | scalar, write mask).

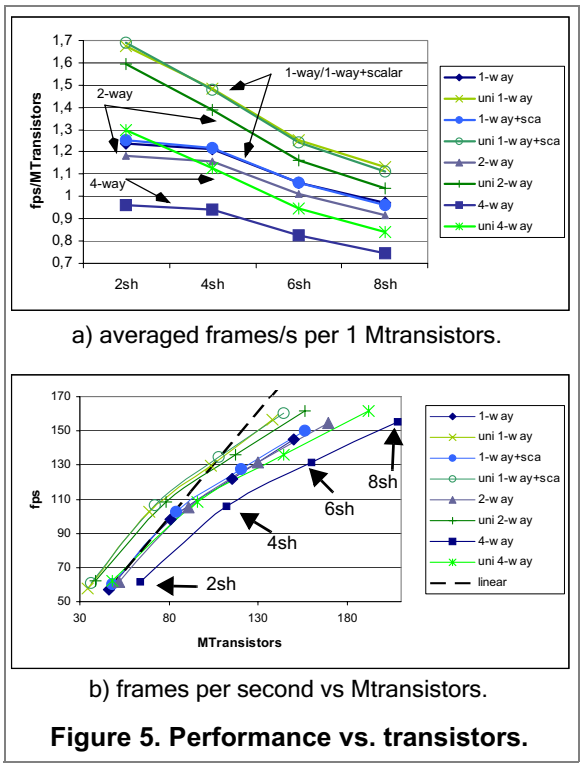
6. Analysis of shader performance

6.1. Unified shader architecture

The experiments in the current section are performed using a 450 frame trace from an Unreal Tournament 2004 Primeval map time demo. The Unreal game engine supports both the OpenGL and Direct3D APIs. The engine (for UT2004) is limited to the OpenGL fixed function API and doesn't use shader programs. However our OpenGL framework generates shader programs that emulate all the fixed function API calls. The vertex and texture load of UT2004 is comparable (or even higher) to other current games.

The vertex shader programs generated for UT2004 are relatively large (in some cases up to 100 instruction long) and implement multiple lights and complex transformations. Meanwhile the fragment programs ar at top a dozen instructions long and implement mostly texture instructions with a few arithmetic instructions combining the texture and input colors and performing the alpha test.

The first 50 and the last 20 frames of the trace aren't used in the experiments because they correspond to load or exit screens. The trace is simulated at a



a) averaged frames/s per 1 Mtransistors.

b) frames per second vs Mtransistors.

Figure 5. Performance vs. transistors.

1024x768 resolution with 8X Anisotropic Filtering enabled. Four regions of 40 frames were rendered for the experiments: frames 100 to 139, 200 to 239, 300 to 339 and 400 to 439. The simulation CPU time (P4 Xeon @ 2 GHz) was 24 hours per region and configuration.

Figure 4 compares the performance of 32 configurations in three axis: unified and non unified shader architectures, number of shader units and the shader instruction fetch/issue width. All the configurations are based on the reference architecture we have described in sections 3 and 5: two ROP units (working on fragment quads) and four 32-bit memory channels. All the non unified shader configurations implement 4 vertex shaders. Figure 4 shows the performance improvement of each configuration relative to the baseline configuration: a non-unified 2 fragment shader 2-way architecture.

As expected, the main performance gain comes from increasing the number of shading units as the inherent parallelism of the independent fragment (or vertices) is exploited. We can clearly see a two fold increase in performance going from 2 to 4 shader units. From 2 to 6 the gain is below linear and it drops further for 2 to 8. The reason, as we will discuss in the next subsections, is that the rendered frame becomes limited by the memory system rather than by fragment shading.

Increasing the fetch/issue width of the shader units allows for up to 8% improvement in some configurations when comparing 1-way to 2-way execution. Going to a 4-way configuration does not yield any additional benefit over the 2-way configuration for our trace set. It must be noted though that the fragment programs in the trace are relatively small and our current OpenGL framework only performs a limited instruction reordering optimization over them. With a better optimizer and larger fragment programs the improvement for the 2-way and 4-way configurations might be higher.

The difference between equivalent configurations for non-unified and unified shader architectures shows that the unified architecture can use the larger shader pool to shade vertices at a faster rate. The improvement is small though, ranging from a 1% to an 8% depending on the configuration. The reason is that the frames in our trace are mostly limited by fragment shading. Another reason is that the same configuration is used for the geometry stages in both architectures and is currently limited to a throughput of 1 vertex and 1 triangle per cycle. The vertex data fetch from memory may also become a bottleneck for not properly aligned or interleaved streams.

6.2. Area comparisons

We have built a rough estimation of the transistor cost of a shader unit based on the difference in transistors between the ATI R400 (160 million transistors for 6 vertex shaders and 4 fragment shader units) and ATI RV400 (120 million transistors for 4 vertex shaders and 2 fragment shader units) as detailed in [5]. Our estimation puts 2.5 million transistors per vertex shader, 15 million transistors per shader unit (fragment or unified) and a 15% increase in transistors per extra SIMD ALU and a 5% increase for the extra scalar ALU. Using this

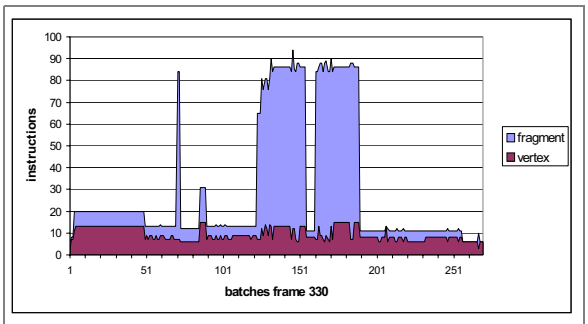


Figure 6. Vertex and fragment program instructions for frame 330.

estimation, Figure 5 compares the same 32 configurations in terms of transistor area and efficiency.

Figure 5 a) compares the configurations in terms or frames per second per million of transistors. Figure 5 b) tallies the simulated frames per second with the esti-

mated area of the configuration. The linear performance line shows a linear improvement of performance per additional transistor based on the non unified 2 shader 1-way configuration. Figure 5 demonstrates that a unified shader architecture is more efficient, ranging from 30% more efficient for the 1-way configurations to 15% more efficient for the 4-way configurations, than a non unified architecture in terms of performance per area. A unified architecture becomes more efficient as the work load (vertices or fragments) of the rendered frame becomes more unbalanced from batch to batch. The UT2004 trace we use is limited in this aspect and other applications may get additional improvements from a unified architecture.

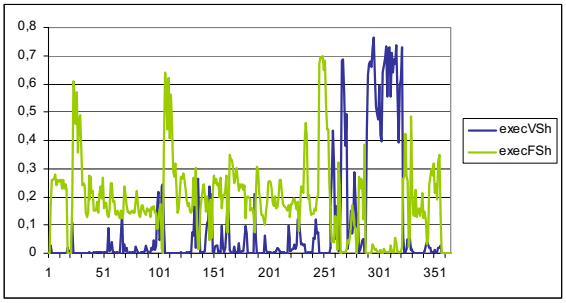


Figure 7. Workload balance in frame 330.

6.3. Detailed performance analysis

In this section we will analyze in detail what are the bottlenecks in the configurations from the previous section. We will use trace frame 330 for the detailed analysis.

Figure 6 shows the number of vertex and fragment program instructions per batch in frame 330 (260 batches are rendered in total). Figure 7 shows the workload balance between the vertex and fragment shader units in the non-unified 8 shader 2-way configuration for frame 330, sampled at 10Kcycle intervals. Batches 120 to 200 (large vertex programs) in Figure 6 correspond with the vertex dominated zone at Figure 7 (cycles 2900K to 3100K). This limited zone accounts for most of the performance difference between the unified and non-unified shader architectures.

Figure 8 shows the average utilization, sampled at 10K cycle intervals, of the three key GPU subsystems: the memory system, the shader execution pipeline and the ROP (color and z) pipeline, Figure 8 also shows the bandwidth consumed for vertex, texture, z and color data for frame 330. The pipeline utilization graphics are normalized to the maximum data, instruction and fragment operation rates. For the memory subsystem, maximum data rate is 1 datum being read or written per cycle; maximum shader execution is 1 instruction per cycle in 8 b) and 2 instructions per cycle in 8 d); maximum ROP bandwidth is 8 fragments/cycle in all configurations. Figures 8 a) and 8 c) are normalized to the maximum per cycle bandwidth also for all the configurations tested so far: 64 bytes per cycle.

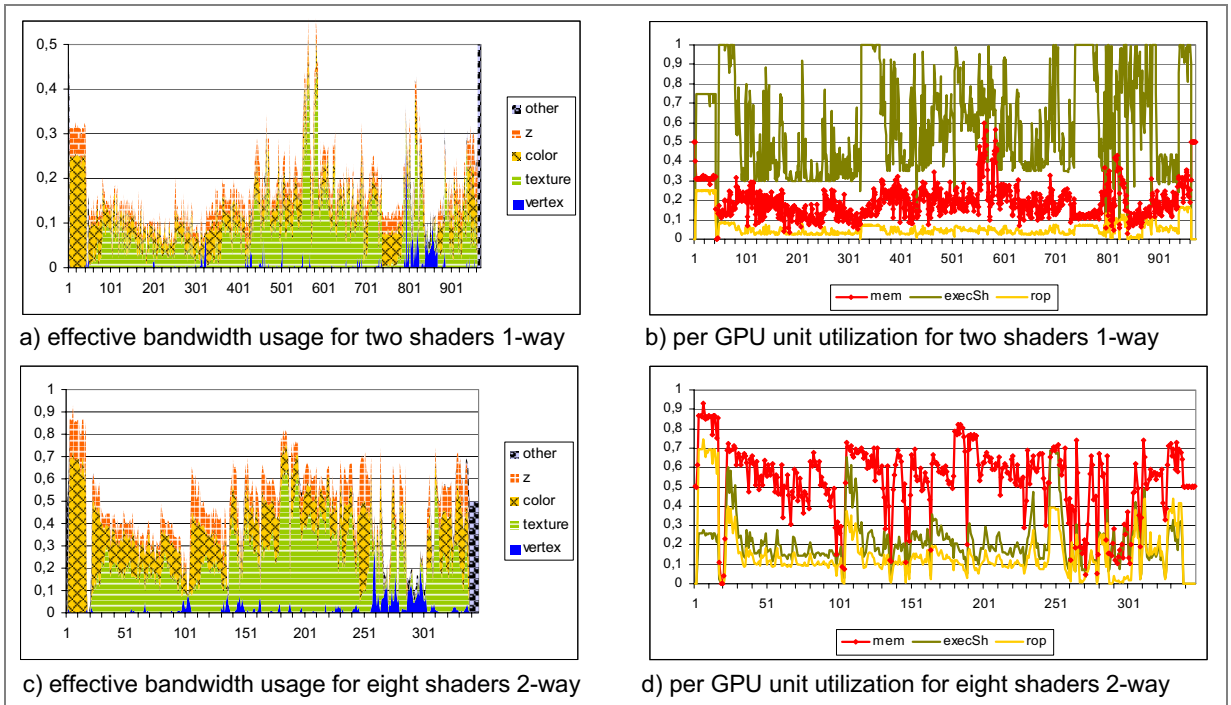
We compare two configurations implementing a unified shader architecture, the first with two shaders units and 1-way fetch/issue width and the second with eight shader units and 2-way fetch/issue width.

Figure 6 shows that most fragment shaders are 6 to 10 instruction long, so given that the ROPs process fragments at a rate of 8 per cycle the GPU would require a pool of 12 to 20 1-way shader units to peak the ROPs. As we can see in figures 8 b) and d) the ROPs only reach a 70% utilization at the start of the frame with the 8 shader configuration. However, even if the two shader configuration is clearly limited by fragment shading the eight shader configuration isn't. Figure 8 d) shows that the average utilization of the shaders is a 30% with peaks of over 50% utilization. If we consider that even for a 2-way shader configuration the shader unit still executes on average 1 instruction per cycle and thread, this 30% is equivalent to a 60% utilization on a 1-way architecture. What keeps the 8 shader configuration from becoming fragment shader limited is the memory system, that as we can se in Figure 8 c) is at 60-70% utilization with a peak for a small fillrate limited zone of 90%. The memory system is unable to go beyond that 60-70% because of inefficiencies and conflicts, and because the increased latency of memory requests can not be hidden by the shader unit threads. Using a pure fill rate benchmark, drawing a full screen quad with forced reading and writing of Z and color buffers, the memory subsystem seems to peak at 80%-90%.

If we analyze the bandwidth usage in Figures 8 a) and c) texture data dominates in both cases, followed by color data and z data. Color data dominates over z data because our architecture supports z compression and fast early rejection using an on chip hierarchical Z buffer, and both techniques save a large amount of z bandwidth. The bandwidth usage for the vertex limited zone of frame 330, see Figures 6 and 7, shows a clear increase in the amount of vertex data read from memory.

6.4. Increasing memory bandwidth

To verify that the memory system is the bottleneck for the six and eight shader configurations we performed an experiment increasing the available band-



a) effective bandwidth usage for two shaders 1-way

b) per GPU unit utilization for two shaders 1-way

c) effective bandwidth usage for eight shaders 2-way

d) per GPU unit utilization for eight shaders 2-way

Figure 8. Frame 330 characterization for two unified shader architectures.

width. If the six and eight shader configuration are unbalanced in terms of bandwidth per shader unit the experiment should show an improvement in performance when the additional bandwidth is provided.

In modern GPUs the number of 32-bit memory channels ranges from 1 in the lower-end segments to 4 in the high-end implementations of the architecture. More than 4 channels add a large number of extra pins in the package and is too expensive to implement with current technology. Therefore for a high end architecture the offered bandwidth can only be modified by changing the memory frequency. Modern GPUs allow running the memory subsystem with a different, faster, clock than the GPU pipeline to provide more than 64 bytes per cycle to the GPU pipeline. However our current implementation of the simulator only supports running the memory subsystem at the same frequency as the GPU pipeline. For this reason we are limited to increasing the number of channels in order to simulate an increase in memory bandwidth.

Figure 9 shows the improvement of adding 1 and 2 additional memory channels, normalized to the non unified two shader 1-way architecture. The bandwidth is increased by 25% and 50% and goes from 64 to 80 and 96 bytes per cycle. The graphic demonstrates that the 6 and 8 shader configurations are limited by memory as the increase in bandwidth produces a a 7% to 14% increase in performance. In this high bandwidth scenario the 8 shader configurations deliver more than a three fold improvement over the base architecture.

7. Related work

NVidia presented their first implementation of a vertex shader for the NV2x GPU architecture [3]. Information obtained from available patents [4] and the analysis of the shaders performance and architecture using shader benchmarks [31] is limited. Some of the information about NVidia and ATI implementations surfaces on unofficial Internet forums [5][6]. Beyond shader microarchitecture, some recent work can be found: T. Aila et al. proposed delay streams [7] and Akenine-Möller described a graphic rasterizer for mobile phones [8]. We can find research on other graphic algorithms: a Reyes renderer was implemented on the Imagine [1] stream processor by Owens [11], the SaarCor [2] group presented a FPGA [13] implementation of their raytracing architecture and ray tracing and photon mapping [12] has been implemented on a modern GPU.

On the side of simulators Stanford has a public software implementation of the OpenGL library that can be used for profiling. GLSim [9] and the GLTrace tool are used in university courses for limited experiments on the graphic pipeline but support for the last OpenGL API specification or extensions is not available. QSil-

ver [10] is a GPU simulator framework that combines a flexible and programmable OpenGL trace capturing and profiling tool, based on the Chromium tool, that feeds a preprocessed trace into a cycle-timer simulation model. QSilver doesn't emulate the GPU functionality and uses statistics and probability distributions as inputs to the model.

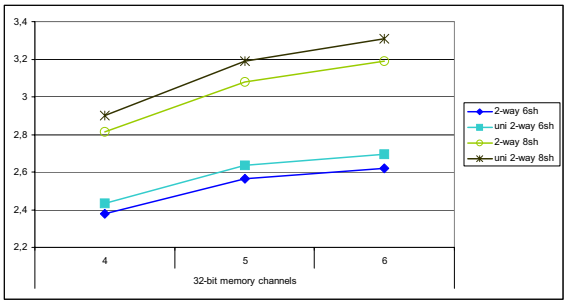


Figure 9. Performance increase with bandwidth increase.

8. Conclusions

Research on the microarchitecture of the shader units in modern GPU has become a hot topic in the last couple of years. Future GPUs will change from the current non-unified shader model towards a unified shader model. This work evaluates the differences in performance and efficiency between both models.

The experiments in this paper show that the main source of performance improvement in modern GPUs comes, as expected, from increasing the number of shader units working in parallel. The increase is near linear as long as the other GPU subsystems, for example memory, don't become the bottleneck. The experiments show that exploiting ILP with superscalar shader units adds an additional 8% performance increase. The unified shader architecture shows a little performance benefit, at least for the tested trace, over the non-unified architecture but the largest gain comes from improved efficiency per area, up to 30% better.

9. References

[1] Ujval Kapasi, et al. The Imagine Stream Processor. Proceedings 2002 IEEE Intl. Conference on Computer Design.

[2] Jörg Schmittler, et al. SaarCOR A Hardware Architecture for Ray Tracing. Graphics Hardware 2002.

[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] Jörg Schmittler, et al. Realtime Ray Tracing of Dynamic Scenes on a FPGA Chip. Graphics Hardware, 2004.

[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 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 Tech.

[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 2002.

[23] S3TC compression: http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt

[24] Stanford University CS488a Fall 2001 Real-Time Graphics Architecture. Kurt Akeley, Path Hanrahan.

[25] Lars Ivar Igesund, Mads Henrik Stavang. Fixed function pipeline using vertex programs. November 22. 2002

[26] Microsoft Meltdown 2003, DirectX Next Slides

[27] ARB Vertex Program extension: http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt

[28] ARB Fragment Program extension: http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt

[29] OpenGL Shading Language v 1.10.

[30] GPGPU. http://www.gpgpu.org/

[31] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. Graphics Hardware 2004.