A Single (Unified) Shader GPU Microarchitecture for Embedded Systems

Victor Moya, Carlos Gonzalez, Jordi Roca

Agustín Fernández, Roger Espasa

Department of Computer Architecture, Universitat Politècnica de Catalunya

Abstract. We present the TILA-rin GPU microarchitecture for embedded systems. We evaluate the architecture using the ATTILA GPU simulation framework. We use a trace from an execution of the Unreal Tournament 2004 PC game to evaluate and compare the performance of the proposed GPU and a baseline GPU architecture for the PC. This work presents the different elements that have been removed from the baseline GPU architecture to accommodate the architecture to the restricted power, bandwidth and area budgets of embedded systems. The unified shader architecture we present processes vertices, triangles and fragments in a single processing unit saving space and reducing hardware complexity. The proposed embedded GPU architecture sustains 25 frames per second on the selected UT 2004 trace.

1 Introduction

In the last years the embedded market has being growing at a fast peace. With the increase of the computational power of the CPUs mounted in embedded systems and the increase in the amount of available memory those systems have become open to new kind of applications. One of those applications are 3D graphic applications, mainly games. Modern PDAs, powerful mobile phones and portable consoles already implement relatively powerful GPUs and support games with similar characteristics and features of PC games from five to ten years ago. However at the current pace embedded GPUs are about to reach the programmability and performance capabilities of their 'big brothers', the PC GPUs. The last embedded GPU architectures presented by graphic companies like NVidia [35], ATI [34] or BitBoys [36] already implement vertex and pixel shaders. The embedded and mobile market are still growing and research on generic and specific purpose processors targeted to these systems will become even more important.

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 [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 also produced an OpenGL driver for our GPU and an OpenGL capture tool able to work in coordination to run full applications (i.e., commercial games) on our GPU microarchitecture. Our microarchitecture and simulator are versatile and highly configurable and can be used to evaluate multiple configurations ranging from GPUs targeted for high-end PCs to GPUs for embedded systems like small PDAs or mobile phones.

We use this capability in the present paper to evaluate a GPU microarchitecture for embedded systems and compare the performance differences from GPU architectures ranging from the high-end PC market to the low end embedded market.

The remainder of this paper is organized as follows: Section 2 describes the graphic pipeline and our baseline GPU microarchitecture. Sections 4 presents the simulator itself and the associated OpenGL framework. In Section 5 we describe and evaluate an implementation of the triangle setup stage in the unified shader for our embedded GPU architecture, a technique useful to reduce the transistors required for a low-end embedded GPU. Section 6 evaluates the performance of our embedded GPU architecture using a trace from the Unreal Tournament 2004 PC game and compares its performance with multiple GPU architecture configurations. Finally Sections 7 and 8 present related work, conclusions and future work.

2 ATTILA Architecture

The rendering algorithm implemented in modern GPUs is based on the rasterization of textured shaded polygons on a color buffer, using a Z buffer to solve the visibility problem. Most GPUs support the rasterization of the most basic polygon, the triangle, as it is more efficiently implementable in hardware. GPUs may optionally include support for line rasterization and the tesselation (conversion to triangles) of quads (planar four vertex polygons) and parametric curved surfaces.

The rendering algorithm can be divided into three main stages: geometry, rasterization and fragment processing. The geometry stage is further divided into the vertex shading and primitive assembly and clipping stages. The rasterization stage is divided into a triangle setup stage and a fragment generation stage, while the fragment processing stage is divided into the fragment shading, fragment tests (Z, stencil and alpha) and color write and blend stages.

Graphic applications use graphics APIs (OpenGL and Direct3D) that implement this rendering pipeline in software or using the capabilities of a GPU.

This section describes our implementation of the rendering pipeline in hardware. Our implementation correlates in most aspects with current real GPUs, except for one design decision: we decided to support from the start a unified shader model [29] in our microarchitecture. The simulator can also be configured to emulate today's hard partitioning of vertex and fragment shaders so that both architectures can be compared.

Figure 1 shows a block diagram of a baseline ATTILA GPU graphic pipeline configured with separated vertex and fragment shaders. We will be referring back to this figure throughout this section. The input and output processing elements of the different ATTILA units (or stages) can be found in Table 1, as well as their bandwidth and queue

sizes. The table shows the configuration for our baseline architecture implementing 4 vertex shaders, two 2-way , 2 ROP units, four 32-bit DDR channels to GPU memory, and a 2 channel bus to system memory (roughly emulating a PCIe x16 bus). Each fragment shader and ROP unit work in this architecture works in a quad (2x2) fragments in parallel. Our architecture can be scaled (down or up) changing the number of shader units and ROP units and their capabilities.

Figure 2 shows the TILA-rin embedded GPU. The embedded architecture is configured with a single unified shader unit and ROP unit that work on a single fragment.

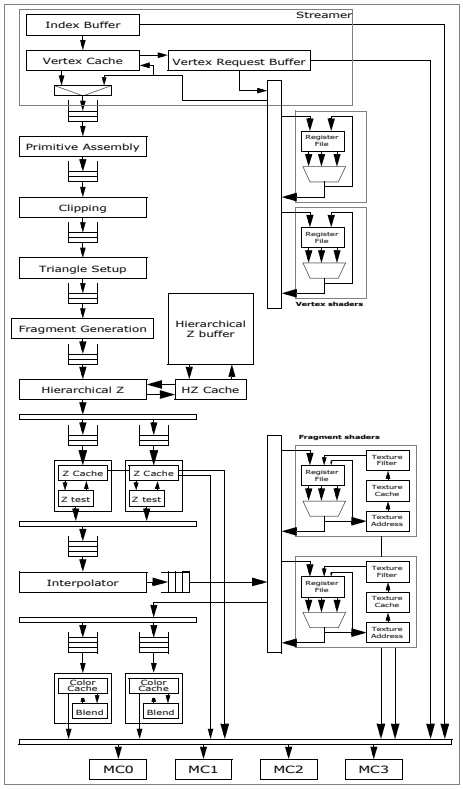

Figure 1 ATTILA Architecture.

All the other GPU units are configured in both cases so they don't become a bottleneck in the most common cases.

2.1 Detailed Pipeline Description

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.

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.

For the embedded GPU architecture proposed in this paper, see Figure 2, the vertex, triangle and fragment inputs are all processed in the same shader unit. Vertex inputs loaded by the Streamer are issued to the shader pool in groups of four (the processing unit in our unified shader units). We must note that the non unified Vertex Shader units work on single vertex inputs rather than on groups of four inputs.

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).

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

Unit	Input Bandwidth	Output Bandwidth	Input Queue		Latency
			Size	Element width	
Streamer	1 index	1 vertex	48	16×4×32 bits	Mem
Primitive Assembly	1 vertex	1 triang.	8	3×16×4×32 bits	1
Clipping	1 triang.	1 triang	4	3×4×32 bits	6
Triangle Setup	1 triang.	1 triang	12	3×4×32 bits	10
Fragment Generation	1 triang	2×64 frag.	16	3×4×32 bits	1
Hierarchical Z	2×64 frag.	2×64 frag.	64	(2×16+4×32)×4 bits	1
Z Test	4 frag.	4 frag.	64	(2×16+4×32)×4 bits	2+Mem
Interpolator	2×4 frag.	2×4 frag.	-	-	2 to 8
Color Write	4 frag.		64	(2×16+4×32)×4 bits	2+Mem
Vertex Shader	1 vertex	1 vertex	12+4	16×4×32 bits	variable
Fragment Shader	4 frag.	4 frag.	240+16	10×4×32 bits	variable

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].

Table 2: Baseline ATTILA architecture caches.

Cache	Size (KB)	Associativity	Lines	Line Size (bytes)	Ports
Texture	**16**	**4**	**256**	**64**	**4x4**
Z	**16**	**2**	**64**	**256**	**4**
Color	**16**	**2**	**64**	**256**	**4**

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 texturesare also supported [23].

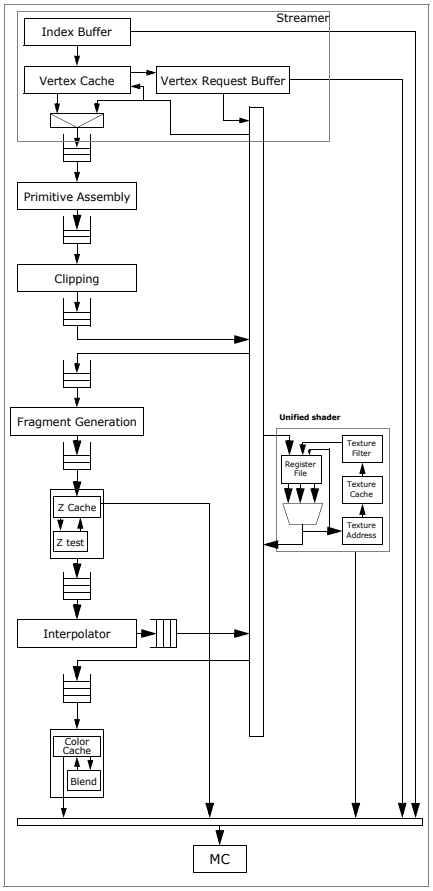

Figure 2 TILA-rin unified shader embedded GPU architecture.

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.

3ATTILA 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).

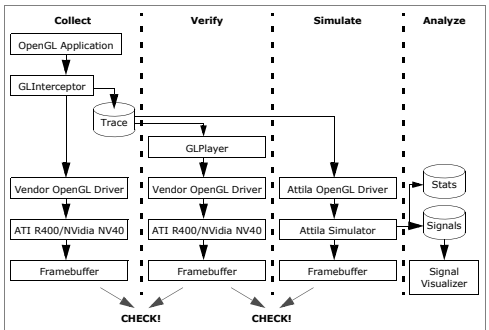

Figure 3 ATTILA Simulation Framework.

Figure 3 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.

4 Unified Shader

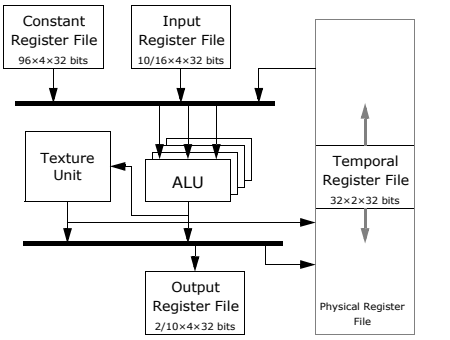

Figure 4 Unified shader architecture

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. SIMD operations like addition, dot product and multiply-add are supported. Scalar instructions (for example, reciprocate, reciprocate square root, exponentiation) operating on a component of a shader register are also implemented. For the fragment shader target and for our unified shader target texture, instructions for accessing memory and a kill instruction for culling the fragment are supported.

The ARB ISA defines four register banks: a read only register bank for the shader input attributes, a write only register bank for the shader output attributes, a read-write temporal register bank to store intermediate results and a read only constant bank that stores data that doesn't change per input but per batch and is shared by all the shader processors. Figure 4 shows the unified shader programming environment.

Shader instructions are stored in a relatively small shader instruction memory (the ARB specification requires a memory for at least 96 instructions) that is preloaded before the batch rendering is started. The shader processor pipeline has a fetch stage, a decode stage, an instruction dependant number of execution stages and a write back stage. The shader processor executes instructions in order and stalls when data dependences are detected. The instruction execution latencies range from 1 cycle to 9 cycles for the arithmetic instructions in our current implementation but can be easily configured in the simulator to evaluate more or less aggressively pipelined ALUs.

Our shader architecture implements multithreading to hide instruction execution latencies and texture access latency exploiting the inherent parallelism of the shader inputs. From the shader point of view all shader inputs are completely independent. A thread represents a shader input. 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. For the fragment shader and unified shader targets more threads are required to hide the latency of the texture accesses therefore we support up to 240 shader inputs on execution in our baseline configuration. A texture access blocks the thread until the texture operation finishes preventing the fetch stage from issuing instructions for that thread.

The number of threads on execution is further limited by the maximum number of temporal registers that the running shader program uses. The ARB ISA defines up to 32 registers in the temporal register bank but less temporal registers are required for the applications that we have tested. We provide a pool of 96 physical registers for the vertex shader target and 480 physical registers for the fragment and unified shader targets.

The vertex programs that we have analyzed until now require 4 to 8 temporal registers per program (vertex programs may have up to a hundred instructions when implementing multiple light sources and complex lighting modes) while the fragment programs require 2 to 4 registers (fragment programs feature no more than a dozen instructions in most analyzed cases).

Another characteristic of our shader model is that we support, for the fragment and unified shader target, to work on groups of shader inputs as a single processing unit. That is a requirement for our current texture access implementation as mentioned in section 2. The same instructions are fetched, decoded and executed for a group of four inputs. For the baseline architecture 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). For the embedded GPU configurations the shader unit executes the same instructions in sequence for one (128 bit wide shader processor) or two inputs (256 bit wide shader processor) from a group until the whole input group is executed. Our fetch stage can be configured to issue one or more instructions for an input group per cycle. Our baseline architecture can fetch and start two instructions for a group per cycle while the embedded architecture can only fetches and starts one instruction per cycle. For the vertex shader target each input is treated independently as vertex shaders in current GPUs are reported to work [3].

4.1 Triangle Setup in the Shader

The triangle rasterization algorithm that our architecture implements (based on [14][15]) targets the triangle setup stage for an efficient implementation on a general purpose CPU [15] or in a separated geometric processor [14]. Our baseline architecture implements the triangle setup stage as a fixed function stage using a number of SIMD ALUs. But the setup algorithm is also suited for being implemented using a shader program. The shader architecture and programming environment is introduced first and then our implementation of triangle setup on a shader is described.

The triangle setup algorithm implemented can be divided in a processing stage that is the part that will be implemented in the unified shader unit (see Figure 2 for the embedded GPU pipeline implementing a single unified shader and triangle setup on the shader) and a small triangle culling stage that will remain as a separated fixed function stage.

The processing stage takes the 2D homogenous coordinate vectors (x, y and w components) for the three vertices that form the triangle and creates an input 3x3 matrix M. Then the adjoint matrix for the input matrix - A = adj (M) - is calculated (lines 1 - 6). The determinant of the input matrix - det(M) - is also calculated (lines 7 - 8). If the determinant is 0 the triangle doesn't generate any fragment and can be culled. The next step of the algorithm described in [14][15] calculates the inverse matrix - M^{-1} = adj(M)/ det(M) - but that step isn't always required and we don't implement it. The resulting setup matrix rows are the three triangle edge equations that are used in the Fragment Generation stage to generate the triangle fragments. The resulting setup matrix can be used to create an interpolation equation to linearly (perspective corrected) interpolate a fragment attribute from the corresponding attribute of the triangle three vertices.

The interpolation equation is computed by a vector matrix product between a vector storing the attribute value for the three vertices and the calculated setup matrix (lines 9

- 11). We only calculate the interpolation equation for the fragment depth attribute as it is the only attribute required for fast fragment depth culling and is automatically generated for each fragment in the Fragment Generator stage. The other fragment attributes will be calculated later in the Interpolation unit using the barycentric coordinates method.

The last step of the processing stage is to adjust the four equations to the viewport size and apply the OpenGL half pixel sampling point offset (lines 12 - 19 and lines 20 - 22). The result of the processing stage are three four component vectors storing the three coefficients (a, b, c) of the edge and z interpolation equations and the a single component result storing the determinant of the input matrix to be used for face culling (lines 23 - 26). Figure 5 lists the shader program used to implement the triangle setup processing stage.

The triangle culling stage culls triangles based on their facing and the current face culling configuration. The triangle facing direction is determined by the calculated input matrix determinant sign. Front facing triangles (for the counter-clock-wise vertex order default for OpenGL) have positive determinants and back facing triangles negative. If front faces are configured to pass triangles are culled when the determinant is negative, if back faces are configured to pass triangles are culled when the determinant is positive. As the Fragment Generator requires front facing edge and z equations, the equation coefficients of the back facing triangle are negated becoming a front facing triangle for the Fragment Generator.

5 Embedded GPU evaluation

Table 3 shows the parameters for the different configurations we have tested, ranging from a middle to high PC GPU to our embedded TILA-rin GPU. The purpose of the different configurations is to evaluate how the performance is affected by reducing the different hardware resources. A configuration could then be selected based on the transistor, area, power and memory budgets for specific target systems. The first configuration shows the expected performance and framerate of the UT2004 game in a high-end PC at a common PC resolution and is used as a baseline to compare the performance of the other configurations and systems. When we reduce the resolution to the typical of embedded systems the same base configuration shows a very high and unrealistic framerate (the game would become CPU limited way before reaching such framerate on any current PC) but it's useful as a direct comparison between the powerful PC configurations and the limited embedded configurations. PC CRT and high-end TFT screens supporting high refresh rates (100+ Hz) are quite common while the small LCD displays of portable and embedded systems only support screen refresh rates in the order of 30 Hz, therefore a game framerate of 20 to 30 is acceptable in the embedded world.

Table 3: Evaluated GPU architecture configurations

Conf	Res	MHz	VSh	(F)Sh	Setup	Fetch way	Regs	Buses	Cache	eDRAM	HZ	Compr
A	1024x768	400	4	2x4	fixed	2	2x480	4	16 KB	-	yes	yes

```
#  Define input attributes
#  Definitions

#
#  Define input attributes
#

ATTRIB iX = triangle.attrib[0];
ATTRIB iY = triangle.attrib[1];
ATTRIB iZ = triangle.attrib[2];
ATTRIB iW = triangle.attrib[3];

#
#  Define the constant parameters
#

#
#  Viewport parameters (constant per batch)
#
#       (x0, y0) : viewport start position
#       (width, height)   : viewport size
#

PARAM rFactor1 = 2.0 / width;
PARAM rFactor2 = 2.0 / height;
PARAM rFactor3 = -(((x0 * 2.0) / width) + 1);
PARAM rFactor4 = -(((y0 * 2.0) / height) + 1);
PARAM offset = 0.5;

#
#  Define output attributes
#

OUTPUT oA = result.aCoefficient;
OUTPUT oB = result.bCoefficient;
OUTPUT oC = result.cCoefficient;
OUTPUT oArea = result.color.back.primary;


#  Define temporal registers
#

TEMP rA, rB, rC, rArea, rT1, rT2;


#  Code

#
#  Calculate setup matrix (edge equations) as
#  the adjoint of the input vertex position
#  matrix.
#

1:  MUL rC.xyz, iX.zxyw, iY.yzxw;
2:  MUL rB.xyz, iX.yzxw, iW.zxyw;
3:  MUL rA.xyz, iY.zxyw, iW.yzxw;
4:  MAD rC.xyz, iX.yzxw, iY.zxyw, -rC;
5:  MAD rB.xyz, iX.zxyw, iW.yzxw, -rB;
6:  MAD rA.xyz, iY.yzxw, iW.zxyw, -rA;
#
#  Calculate the determinant of the input
#  matrix (estimation of the signed 'area'
#  for the triangle).
#

7:  DP3 rArea, rC, iW;
8:  RCP rT2, rArea.x;

#
# Calculate Z interpolation equation.
#

9:  DP3 rA.w, rA, iZ;
10:  DP3 rB.w, rB, iZ;
11:  DP3 rC.w, rC, iZ;

#
#  Apply viewport transformation to equations.
#

12:  MUL rT1, rA, rFactor3;
13:  MAD rT1, rB, rFactor4, rT1;
14:  ADD rC, rC, rT1;
15:  MUL rA, rA, rFactor1;
16:  MUL rB, rB, rFactor2;
17:  MUL rA.w, rA.w, rT2.x;
18:  MUL rB.w, rB.w, rT2.x;
19:  MUL rC.w, rC.w, rT2.x;

#
#  Apply half pixel sample point offset
#  (OpenGL).
#

20:  MUL rT1, rA, offset;
21:  MAD rT1, rB, offset, rT1;
22:  ADD rC, rC, rT1;

#
#  Write output registers.
#

23:  MOV oA, rA;
24:  MOV oB, rB;
25:  MOV oC, rC;
26:  MOV oArea.x, rArea.x;
27:  END;
```

Figure 5 Shader program for Triangle Setup.

Table 3: Evaluated GPU architecture configurations

Conf	Res	MHz	VSh	(F)Sh	Setup	Fetch way	Regs	Buses	Cache	eDRAM	HZ	Compr
B	320x240	400	4	2x4	fixed	2	2x480	4	16 KB	-	yes	yes
C	320x240	400	2	1x4	fixed	2	1x480	2	16 KB	-	yes	yes
D	320x240	400	2	1x4	fixed	2	1x480	2	8 KB	-	no	yes
E	320x240	200	-	2	fixed	2	240	1	8 KB	-	no	yes
F	320x240	200	-	2	fixed	2	240	1	4 KB	-	no	no

Table 3: Evaluated GPU architecture configurations

Conf	Res	MHz	VSh	(F)Sh	Setup	Fetch way	Regs	Buses	Cache	eDRAM	HZ	Compr
G	320x240	200	-	1	fixed	2	120	1	4 KB	-	no	no
H	320x240	200	-	1	fixed	1	120	1	4 KB	-	no	no
I	320x240	200	-	1	on shader	1	120	1	4 KB	-	no	no
J	320x240	200	-	1	on shader	1	120	1	4 KB	1 MB	no	no
K	320x240	200	-	1	on shader	1	120	1	4 KB	1 MB	yes	yes

The configurations A and B roughly correspond to the configurations of ATI Radeon 9800 (R350) and ATI Radeon X700 (RV410) graphic cards, while the configuration C could correspond to an ATI Radeon X300 (RV370). The other configurations range from the equivalent of a GPU integrated in the PC chipset or a high end PDA to a low end embedded GPU. Configurations A to I have 128 MB of GPU memory (UT2004 requires at least 64 MBs for textures and buffers) and can access up to 64 MBs of system memory. Configurations J and K have 1 MB of embedded DRAM for the framebuffer (GPU memory) and can access up to 128 MBs of system memory.

Table 3 shows the following parameters: the resolution (Res) at which the trace was simulated, an estimated working frequency in MHz for the architecture (400 MHz or more for a PC GPU is already common for the middle and high-end segments while 200 MHz is in the middle to high-end segment of the current low power embedded GPUs); the number of vertex shaders for non unified shader configurations (VS) and the number of fragment shaders (FSh) for non unified shader configurations or the number unified shaders (Sh) for unified shader configurations. The number of fragment shaders is specified by the number of fragment shader units working on whole fragment groups in parallel multiplied by the number of fragments in a group (always 4). For the unified shader configurations the number represents how many shader inputs (vertices, triangles or fragments) from an input group in the shader are processed in parallel. The Setup parameter indicates if the Triangle Setup processing stage is performed in a separated fixed function ALU or in the shader. The Fetch way parameter is the number of instructions that can be fetched and executed per cycle for a shader input or group. The number of registers represents the total number of temporal registers available in the fragment or unified shader units for all threads in execution (the maximum number of threads that can be in execution at any time is half the number of registers since at least two temporal registers are always assigned per thread). The number of 16 bytes/cycle channels used to access the GPU memory (or the embedded memory for configurations J and K) can be found in the column labeled Buses. The size of the caches in the GPU is defined by the Caches parameter and the eDRAM parameter defines how many embedded DRAM or reserved SRAM (on the same die or in a separate die) is available for storing the framebuffer. Finally the HZ and Compr. parameters show if the configuration implements Hierarchical Z and Z compression. When Hierarchical Z and Compression are enabled the Z and Color cache lines are 256 bytes wide (storing a block of 8x8 fragments) but when they are disabled the cache line size is 64 bytes (4x4 fragments).

5.1 Benchmark Description

We selected a 150 frame trace from an Unreal Tournament 2004 map for the evaluation of our embedded architecture. UT2004 is a game that uses a version of the Unreal game engine supporting both the OpenGL and Direct3D APIs. The UT2004 version of the Unreal engine uses the fixed function vertex and fragment OpenGL pipelines, and our OpenGL implementation generates, as is done for real GPUs, our own shader programs from the fixed function API calls, as discussed in section 4. In terms of the shader programs we generate for UT2004 the vertex programs are relatively large (reaching almost 100 instructions) implementing multiple lights and complex transformations while the fragment programs are mostly texture accesses with a few instructions combining the colors and performing alpha testing. However the vertex and texture load of UT2004 is comparable (or even higher) to current games.

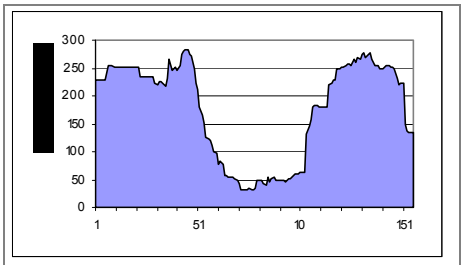

Figure 6 Vertices per frame for the UT2004 trace

We don't currently support tracing OpenGL ES (OpenGL for embedded and mobile systems) applications so we had to use a trace from the PC world. The workload of the Unreal trace, even at lower resolutions, may be quite heavyweight (in terms of texture sizes, memory used and vertex load) compared with current embedded graphic applications but we consider that is a good approximation for future embedded graphic applications. The trace was generated at a resolution of 640x480 in a Pentium4 2.8 GHz PC with a GeForce 5900. For the experiments we modified the trace resolution (OpenGL call glViewport()) to use the 1024x768 (PC standard resolution) and 320x240 (embedded standard resolution) resolutions. Figure 6 shows the vertex load per frame for the selected trace. The trace shows a player running around a forest with large woods and high cliffs and the middle region with fewer vertices corresponds with a clearing in the woods. We didn't simulate the whole trace but 20 frames at four representative regions: frame 30 to 50, frame 60 to 80, frame 100 to 120 and frame 120 to 140. For the configuration A (largest resolution and size of the simulated architecture) the simulation of each region lasted around 10 hours for 100 million simulated cycles, for the other configurations the simulation lasted up to 5 hours and 180 million cycles (on a P4 Xeon @ 2 GHz).

5.2 Performance

Figure 7 shows the simulated framerate for the different configurations. Figure 8 compares the peak bandwidth (GB/s), Gflops and the total amount of cache memory for each configuration. Figure 9 shows the relation between peak bandwidth, peak computation and the amount of cache memory and the performance rendering the selected trace.

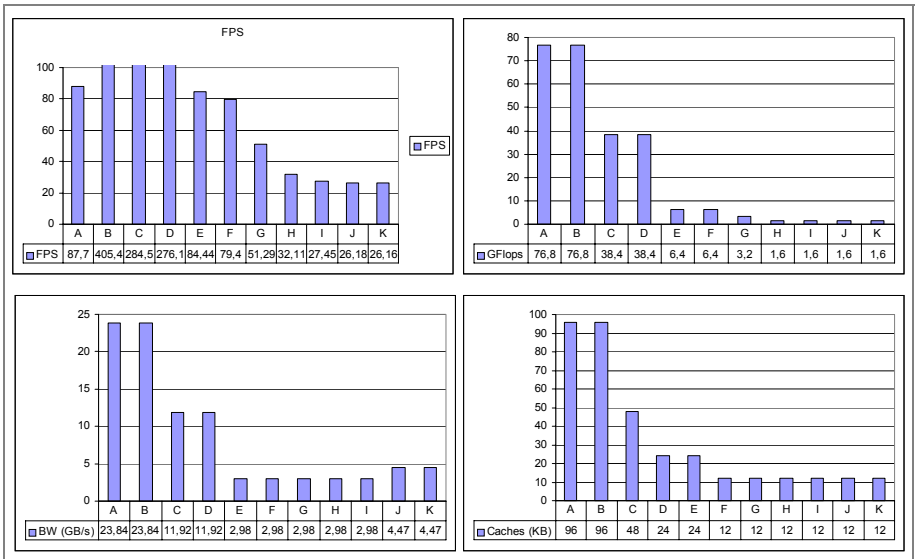

Figure 7 FPS, peak bandwidth, peak shading power and total cache memory for all the GPU configurations.

Figure 9 clearly shows that the performance per computational power in the shader units increases as both the bandwidth and the computational power are reduced. This may signal that the GPU architecte for the selected trace is limited by the memory subsystem. Our current implementation of the Memory Controller isn't efficient enough to provide the shader and other stages of the GPU pipeline with enough data with the available bandwidth. Improving and increasing the accuracy of the simulated memory model will be a key element of our future research.

Figure 7 shows that the configurations J and K that implement a separated memory pool for the framebuffer (depth and color) and access the texture and vertex buffer data from the system memory has worst performance than configuration I that keeps all the data in GPU specific memory. One possible reason is that the main consumer of memory in the UT2004 trace is texture access and not framebuffer updates (as it may be common in other graphic applications). As in configuration I the textures are accessed using a fast bus (4 cycles latency) providing 16 bytes per cycle and in configuration J and K they are accessed through a slower 8 bytes/cycle bus with quite higher latency (50 cycles) that can not be properly hide by the number of threads supported in the shader unit. However configuration J and K may be more representatives of a low end embedded GPU than configuration I.

When enabling Hierarchical Z and framebuffer compression in configuration K there is slight performance reduction rather than the expected improvement. The reason is related to the small cache sizes that can't support the largest rasterization tiles and cache lines that our current implementation of Hierarchical Z and compression required. We may change in the future to implemente versions of both algorithms for smaller cache lines.

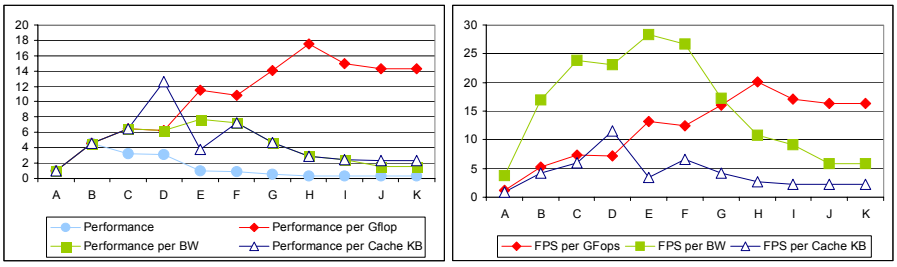

Figure 8 The graphic in the left shows the relative performance per unit resource (1 GFlop, 1 GB/s and 1 KB). The graphic in the right shows the frames per second for the same resources

5.3 Triangle Setup in Shader Performance Analysis

To test the overhead of removing the fixed function unit that performs the processing stage of triangle setup we use three different traces. The *torus* trace renders four colored torus lit by a single light source. This trace has a heavy vertex and triangle load and a limited fragment load (no fragment shader is used). The *VL-II* trace renders a room with a volumetric lighting effect implemented with a 16 instruction fragment shader. The vertex/triangle load for this trace is low. The third trace is the same Unreal Tournament 2004 trace that we have used in the previous subsection. For the test we use the embedded GPU architecture configurations H and I that can be found in Table 3.

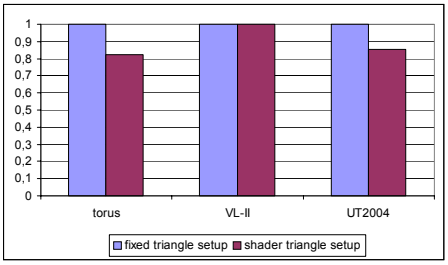

Figure 9 Performance of triangle setup on shader

Figure 10 shows the relative performance of the H (triangle setup in the fixed function unit) and I (triangle setup in the shader unit) configurations for the three traces. The overhead of performing triangle setup in the shader increases as the triangle load (usually at a ratio of 1:1 or 3:1 -depending on the rendering primitive- with the vertex load) increases in relation with the fragment load and we can see that for the *torus* trace there is a 20% reduction in performance between the configuration with the fixed function unit and the one performing the processing in the shader. For fragment limited applications like the *VL-II* demo the performance reduction is too small to be noticeable. The UT2004 game is a more complex graphic application and we can find frames and batches that are fragment limited and others that have a high vertex/triangle load (see Figure 6). UT2004 has a relatively high vertex/triangle load on average and as we can see the performance reduction is in the order of a 15%, not as low as the *torus* trace but still relatively high.

Performing triangle setup on the shader will make sense when the transistor budget for the GPU architecture is low and removing the large fixed function ALU can save area and power, while GPU performance remains acceptable for the targeted appliations. The embedded GPU configuration is clearly limited by shading performance (only one shader unit executing a single instruction per cycle). Performing Triangle Setup in the shader may be also useful when shading performance is so high that the performance overhead becomes small, in the order of a 1% reduction, for example in a PC unified architecture with 8+ shaders.

6 Related Work

A complete picture of the architecture of a modern GPU is difficult to acquire just from the regular literature. However NVidia presented their first implementation of a vertex shader for the GeForce3 (NV2x) GPU [3] and information from available patents [4], even if limited, complemented with the analysis of the performance of the shader units [ref GPU Bench, and others] in current GPUs provide useful information. More actuallized information about NVidia and ATI implementations surfaces as unofficial or unconfirmed information on Internet forums [5][6]. Outside shader microarchitecture some recent works can be found. For example, T. Aila et al. proposed delay streams [7], as a way to improve the performance of immediate rendering GPU architectures with minor pipeline modifications.

Research papers and architecture presentations about embedded GPU architectures have become common in the last years. Akenine-Möller described a graphic rasterizer for mobile phones [8] with some interesting clues about how the graphic pipeline is modified to take into account low power and low bandwidth, while keeping an acceptable performance and rendering quality. Bitboys [36] and Falanx [37], both small companies working on embedded GPU architectures, presented their future architectures and their vision of the importance of embedded GPU market in Hot3D presentations in Graphics Hardware 2004 conference.

7 Conclusions

Our flexible framework for GPU microarchitecture simulation allows to accurately model and evaluate a range of GPU configurations from the high-end PC GPU to the low-end embedded GPU. We have presented a low budget embedded GPU with a single shader unified shader that performs triangle setup on the shader while keeping a reasonable performance for embedded graphic applications.

We have evaluated the performance of different configurations of our generic GPU architecture ranging from a low budget PC GPU architecture to the proposed embedded GPU architecture. We have evaluated to cost of performing triangle setup in the unified shader unit.

The proposed embedded architectures achieves with a single unified shader and ROP pipe a sustainable rate of 25 frames per second with the selected Unreal Tournament trace at a resolution of 320x240.

References

[1] Ujval Kapasi, et al. The Imagine Stream Processor. Proceedings 2002 IEEE International 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 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] Volumetric Lighting II. Humus 3D demos: http://www.humus.ca/

[33] PowerVR MBX http://www.powervr.com/Products/Graphics/MBX/Index.asp

[34] ATI Imageon 2300 http://www.ati.com/products/imageon2300/features.html

[35] NVidia GoForce 4800 http://www.nvidia.com/page/goforce_3d_4500.html

[36] Bitboys G40 Embedded Graphic Processor. Hot3D presentations, Graphics Hardware 2004.

[37] Falanx Microsystems. Image Quality no Compromise. Hot3D presentations, Graphics Hardware 2004.