A Single (Unified) Shader GPU Microarchitecture for Embedded Systems

Victor Moya[1], Carlos González, Jordi Roca

Agustín Fernández, Roger Espasa[2]

Department of Computer Architecture, Universitat Politècnica de Catalunya

Abstract. We present and evaluate the TILA-rin GPU microarchitecture for embedded systems 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 embedded GPU against a baseline GPU architecture for the PC. We evaluate 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 20 frames per second on the selected UT 2004 trace.

1 Introduction

In the last years the embedded market has been 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 these 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 PowerVR [18], NVidia [19], ATI [20] or BitBoys [21] 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.

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.

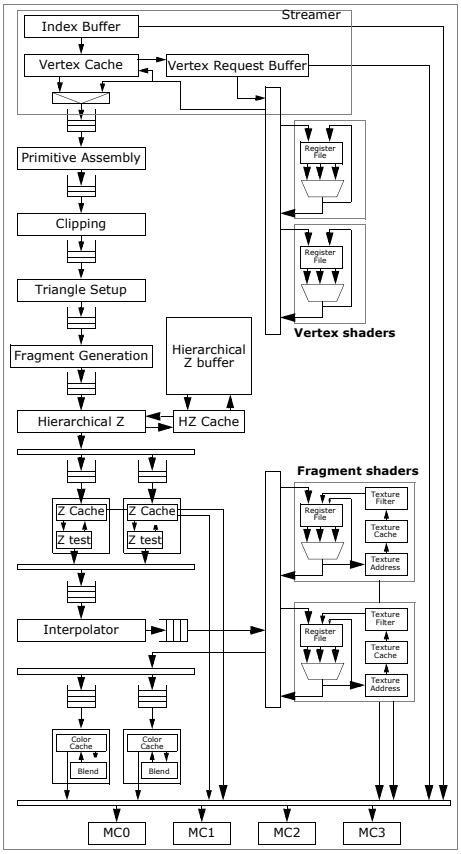

Figure 1 ATTILA Architecture.

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 [12], 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 middle-end PC market to the low end embedded market.

The remainder of this paper is organized as follows: Section 2 introduces the ATTILA microarchitecture and compares the baseline GPU pipeline with the proposed TILA-rin embedded GPU pipeline. Sections 3 presents the simulator itself and the associated OpenGL framework. Section 4 describes our unified shader architecture and our implementation of the triangle setup stage in the unified shader, a technique useful to reduce the transistors required for a low-end embedded GPU. Section 5 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 6 and 7 present related work and conclusions.

2 ATTILA Architecture

The rendering algorithm implemented in modern GPUs is based on the rasterization of textured shaded triangles on a color buffer, using a Z buffer to solve the visibility

problem. GPUs implement this algorithm with the pipeline shown at figure 1 and 2: the streamer stage fetches vertex data, the vertex shader processes vertices, the primitive assembly stage groups vertices into triangles, the clipper culls non visible triangles, the triangle setup and fragment generator stages create fragments from the input triangle, the z stage culls non visible fragments, the fragment shader processes those fragments and the color stage updates the framebuffer. Graphic applications can access and configure the GPU pipeline using graphic APIs like OpenGL and Direct3D.

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

Unit	Input Bandwidth	Output Bandwidth	Input Queue		Latency in cycles
			Size	Element width	
Streamer	1 index	1 vertex	80	16×4×32 bits	Mem
Primitive Assembly	1 vertex	1 triang.	64	3×16×4×32 bits	1
Clipping	1 triang.	1 triang	32	3×4×32 bits	6
Triangle Setup	1 triang.	1 triang	32	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.	112+16	10×4×32 bits	variable

Our implementation of the pipeline correlates in most aspects with current real GPUs, except for one design decision: we decided to support from the start a unified shader model [14] in our microarchitecture. The simulator can also be configured to emulate today's hard partitioning of vertex and fragment shaders so that both models can be compared.

Table 2: Baseline ATTILA architecture caches.

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

Figure 1, Table 1 and Table 2 describe our baseline non unified architecture: 4 vertex shaders, two 2-way fragment shaders, 2 ROP (raster operator) units, four 64-bit DDR channels to GPU memory, and a 2 channel bus to system memory (similar to PCI Express). Our architecture can be scaled (down or up) changing the number of shader and ROP units and their capabilities. Table 1 shows the input and output processing elements for each pipeline stage, as well as their bandwidth, queue sizes and latencies in cycles. Table 2 shows the configuration of the different caches in the pipeline. Figure 2 shows the TILA-rin embedded GPU. The embedded architecture is configured

with a single unified shader and ROP unit. A more detailed description of the ATTILA pipeline and stages can be found at [23]. In this section we will only compare the baseline architecture and the proposed embedded architecture.

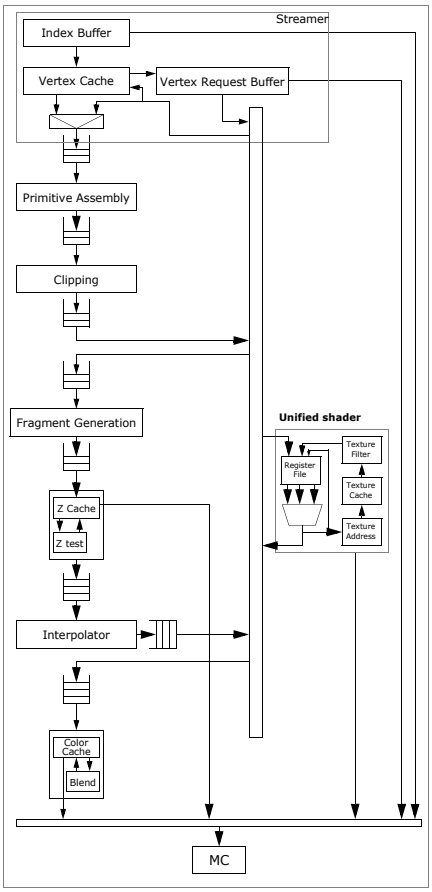

Figure 2 TILA-rin unified shader embedded GPU architecture.

Comparing Figures 1 and 2 we can see the main differences. The baseline PC GPU configuration implements separated vertex (four, but only two shown in Figure 1) and fragment shader units while the TILA-rin embedded GPU implements a single unified shader. This single unified shader processes three kind of inputs: vertices, triangles and fragments.

The Triangle Setup stage is mostly removed (further discussed at section 4.1) from the embedded GPU and the triangle setup computation is performed in the single shader unit. The on die Hierarchical Z buffer [10][11] and associated fast fragment cull stage are also removed in the embedded GPU to reduce the transistor budget.

The embedded architecture implements a single ROP pipeline (z test, stencil test and color write), two in the baseline architecture, and a single channel to GPU memory (16 bytes per cycle) compared with four in the baseline architecture. The bus to system memory is the same for both configurations. Z compression isn't supported in the embedded architecture and the Z and color cache are smaller, 4 KB, with the cache line size set at 64 bytes rather than 256 bytes (Table 2).

The ROP stages also differ in the rate at which fragments are processed. While the embedded architecture keeps a quad (2x2 fragments) as the work unit a reduced number of ALUs, internal bus widths and bandwidth limit the throughput to one fragment per cycle. The baseline PC ROP units can process a whole quad in parallel and output one quad per cycle.
The baseline fragment shader unit can also process a whole quad in parallel while the vertex shader and unified shader only work on a single processing element per cycle.

3 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 con-

figurable (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 buses between the simulated stages. A stage uses the data received from its input buses 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.

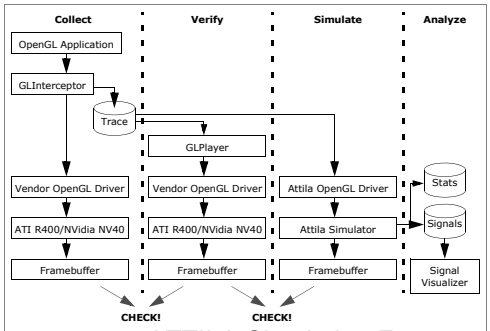

Figure 3 ATTILA Simulation Framework.

Our simulator implements a "hot start" technique that allows the simulation to be started at any frame of a trace file. Frames, disregarding data preloaded 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 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. The stack top layer manages the API state while the lower layer offers a basic interface for configuring the hardware and allocating memory. The current implementation supports all the API features required to fully simulate a graphic application. One key feature is the conversion of the vertex and fragment fixed function API calls into shader programs [13], required applications using the pre programmable shading API.

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 3. All this information is stored in an output file (a trace). To verify the integrity and faithfulness of the recorded trace the 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

Our unified shader architecture is based on the ISA described at the ARB vertex and fragment program OpenGL extensions [15][16]. 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 single 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.

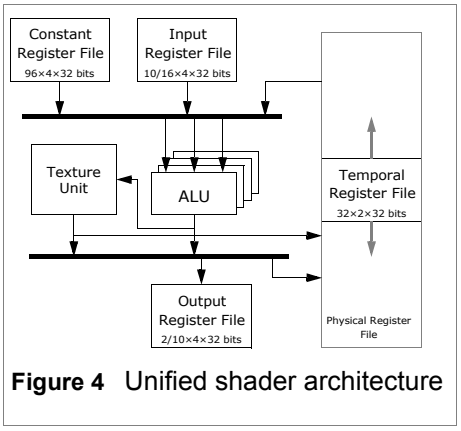

Figure 4 Unified shader architecture

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 112 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 live registers that the running shader program requires. 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 448 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. This is a requirement for our current texture access implementation as explained in [23]. 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 bits 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 fetch and start one instruction per cycle. For the vertex shader target each input is treated independently as vertex shaders in current GPUs are reported to work [1].

4.1 Triangle Setup in the Shader

The triangle rasterization algorithm that our architecture implements (based on [8][9]) targets the triangle setup stage for an efficient implementation on a general purpose CPU [9] or in a separated geometric processor [8]. 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 triangle setup algorithm implemented can be divided in a processing stage that is the part that will be implemented in the unified shader unit (Figure 2) and a small triangle culling stage that will remain as a separated fixed function stage. Figure 5 shows the shader program used to perform the setup computation for an input triangle.

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 [8][9] 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 half plane 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 perform linear perspective corrected interpolation of the fragment attributes from the corresponding triangle vertex attributes.

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

```
#  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.

- 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 20 - 22). The result of the processing stage are three 4 component vectors storing the three coef-

ficients (a, b, c) of the triangle half plane edge and z interpolation equations and a single component result storing the determinant of the input matrix to be used for face culling (lines 23 - 26).

The triangle culling stage culls triangles based on their facing and the current face culling configuration. The sign of the input matrix (M) determinant determines the facing direction of the triangle. 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.

Table 3: Evaluated GPU architecture configurations

Conf	Res	MHz	VSh	(F)Sh	Setup	Fetch way	Reg Thread	Buses	Cache	eDRAM	HZ	Compr
A	1024x768	400	4	2x4	fixed	2	4x112	4	16 KB	-	yes	yes
B	320x240	400	4	2x4	fixed	2	4x112	4	16 KB	-	yes	yes
C	320x240	400	2	1x4	fixed	2	4x112	2	16 KB	-	yes	yes
D	320x240	400	2	1x4	fixed	2	4x112	2	8 KB	-	no	yes
E	320x240	200	-	2	fixed	2	2x112	1	8 KB	-	no	yes
F	320x240	200	-	2	fixed	2	2x112	1	4 KB	-	no	no
G	320x240	200	-	1	fixed	2	4x60	1	4 KB	-	no	no
H	320x240	200	-	1	fixed	1	4x60	1	4 KB	-	no	no
I	320x240	200	-	1	on shader	1	4x60	1	4 KB	-	no	no
J	320x240	200	-	1	on shader	1	4x60	1	4 KB	1 MB	no	no
K	320x240	200	-	1	on shader	1	4x60	1	4 KB	1 MB	yes	yes

5 Embedded GPU evaluation

Table 3 shows the parameters for the different configurations we have tested, ranging from a middle level 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 middle-end PC at a common PC resolution and is used as a baseline to compare the performance of the other configurations. 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) 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.

The tests cases A and B correspond to the same configuration, equivalent to an ATI Radeon 9800 (R350) or 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 a PC chipset or a high end PDA to a low end embedded GPU. Configurations A to I implemente 128 MBs of GPU memory (UT2004 requires more than 64 MBs for textures and buffers) and 64 MBs of higher latency system memory. Configurations J and K are configured with 1 MB of GPU memory for the framebuffer (could be implemented as embedded DRAM), with a maximum data rate of 16 bytes/cycle, and 128 MBs of higher latency lower bandwidth system memory.

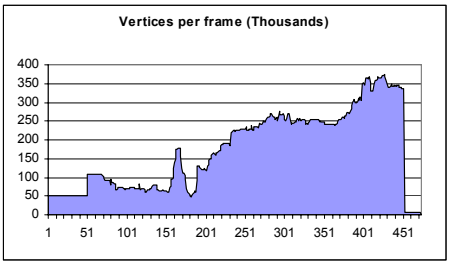

Figure 6 Vertices per frame for the UT2004 trace

Table 3 shows the following parameters: resolution (Res) at which the trace was simulated, an estimated working frequency in MHz for the architecture (400+ MHz is common for the middle and high-end PC GPU segments, 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 of unified shaders (Sh) for unified shader configurations. The number of fragment shaders is specified by the number of fragment shader units 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 a four input group 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 registers per thread parameter (Regs/Thread) represents the number of threads and temporal registers per thread for each shader unit in the GPU. 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 disabled Z and color cache line size becomes 64 bytes).

We selected the different configurations reducing, from the baseline configuration A, the number of shader and ROP pipelines and, at the same time, reducing the peak bandwidth as it exceeds the configuration requeriments. We reduce therefore the cost in area and the cost of the memory. We reduce the frequency to reduce power (the required voltage also goes down with the frequency). We reduce the cache sizes when the ratio of cache area versus ALU area changes. After reaching the single unified shad-

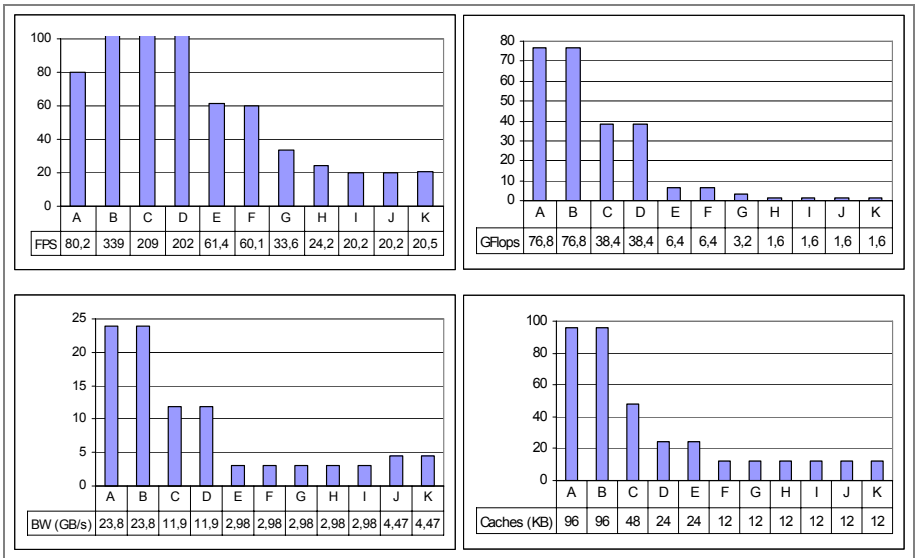

Figure 7 FPS, peak bandwidth, peak shading power and total cache memory for all the GPU configurations.

er configuration (G) we keep reducing the area requeriment, first removing the second SIMD ALU (H) from the shader, then removing the triangle setup unit and performing setup in the shader (I). We finally test the impact of keeping the texture data in system memory (higher latency and half the bandwidth than the GPU memory) while using 1 MB of GPU memory for the framebuffer (J). For this configuration we also test the effect of implementing bandwidth saving techniques: Hierarchical Z and Z compression (K).

5.1 Benchmark Description

We have selected a 450 frame trace from a timedemo of the Unreal Tournament 2004 Primeval 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 API, 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 3. The generated vertex programs are relatively large (20 to 90 instructions) while the fragment programs are mostly texture accesses with a few instructions combining the colors and performing alpha test (6 to 15 instructions). However the vertex and texture load of UT2004 is comparable (or even higher) to other current games.

We don't currently support tracing OpenGL ES (OpenGL for embedded and mobile systems) applications so we use a trace from a PC game. The workload (texture sizes, memory used and vertex load) of the selected Unreal trace, even at lower resolutions, is quite heavyweight 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 800x600 in a Pentium4 at 2.8 GHz PC with a GeForce

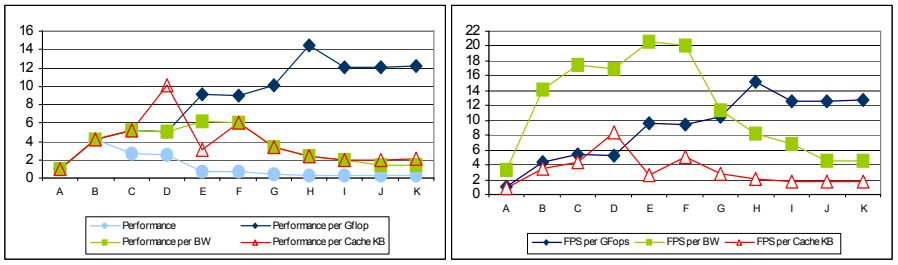

Figure 8 The graphic in the left shows the relative performance per unit resource (1 GFlop, 1 GB/s and 1 KB). The graphic in the right shows the frames per second for the same resources

5900. For the experiments we modified the trace resolution (OpenGL call glViewport()) to use the 1024x768 (PC) and 320x240 (embedded) resolutions. Figure 6 shows the vertex load per frame for the selected trace. We didn't simulate the whole trace but six representative regions: frames 100 to 120, 200 to 220, 250 to 270, 300 to 320, 350 to 370 and 400 to 440. For configuration A (highest workload) the simulation lasted 7 to 14 hours for, depending on the region, 100 to 240 million simulated cycles. For the other configurations the simulation lasted from 1 to 7 hours for 150 to 500 million cycles depending on the configuration and simulated region. The simulations were performed on an 80 node cluster of P4 Xeon @ 2 GHz.

5.2 Performance

Figure 7 shows the simulated framerate (frames per second or FPS), peak bandwidth (GB/s), peak shader GFlops and the total amount of cache memory for each configuration. Figure 8 shows the relation between peak bandwidth, peak computation and the total cache memory and performance (measured in frames per second) normalized to test case A.

Figure 8 shows that the efficiency of shader processing (GFlops) decreases (configurations H to A) as the number of shader units and ALUs per shader unit increases. These configurations are likely 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 configured bandwidth. Improving and increasing the accuracy of the simulated memory model will be a key element of our future research. Another source of inefficiency for the 2-way configurations is the reduced ILP of the shader programs used in the trace.

Comparing, in Figure 7, configuration I against configurations J and K there is no noticeable performance hit which seems to indicate that for configuration I GPU memory bandwidth exceeds the shader processing and latency hiding capacity. Implementing a lower cost memory (configurations J and K or even less expensive configurations) makes sense for this embedded configuration. In Figure 8 we confirm that the efficiency per GB/s of bandwidth provided is higher for configurations E and F and becomes worse for all other configurations, signaling that the configured shader capacity and

memory bandwidth may not optimal for the selected trace. Other graphic applications (less limited by shader processing or using bandwidth consuming techniques, for example antialiasing) may show different shading to bandwidth requirements.

Enabling Hierarchical Z and framebuffer compression in configuration K has no noticeable effect on performance. The reason is that the 1 MB framebuffer memory provides more than enough bandwidth for the framebuffer accesses and bandwidth saving techniques are not useful.

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 five different traces. The *torus* trace renders four colored torus lit by a single light source. The trace has a heavy vertex and triangle load and a limited fragment load (minimum fragment shader used). The *lit spheres* trace renders four spheres with multiple light sources active. The vertex and triangle workload is high, the vertex shader program is large and the fragment shader program is the minimum (copy interpolated color). The *space ship* trace renders a multitextured space ship for a high vertex workload and small vertex and fragment shader programs. The trace may be limited by vertex data reading. The *VL-II* trace [17] renders a room with a volumetric lighting effect implemented with a 16 instruction fragment shader. The vertex and triangle load for this trace is low. The fifth trace is the Unreal Tournament 2004 trace (same simulated regions) that we have used in the previous section.

Figure 9 shows the relative performance of configurations H (triangle setup in specific hardware) and I (triangle setup in the shader unit) for the five selected traces. The overhead of performing triangle setup in the shader increases as the triangle load (usually at a ratio 1:1 to 3:1 with the vertex load, depending on the rendering primitive and post shading vertex cache hits) increases in relation with the fragment load. Figure 9 shows that for the *torus* trace there is a 16% hit on performance when performing triangle setup in the single shader unit. For a fragment limited application like the *VL-II* demo the performance reduction is too small to be noticeable. In mixed workload applications or applications limited by other stages (vertex shading for *lit spheres* and vertex data bandwidth for the *space ship*) the performance hit is reduced. 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). But for the tested rendering resolution, 320x240, UT2004 has a relatively high vertex and triangle workload on average and we can see that the performance hit is quite noticeable: a 16% performance reduction. From the results in the previous subsection we also know that the UT2004 trace is limited by shader processing

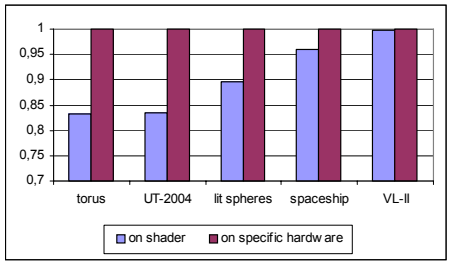

Figure 9 Cost of performing triangle setup in the shader

in both configurations and increasing the shader load, shading triangles, helps to increase the performance hit.

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 keeping an acceptable performance for the targeted applications. 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+ shader units.

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 [1] and information from available patents [2], even if limited, complemented with the analysis of the performance of the shader units [24][25] in current GPUs provides an useful insight in the architecture of a modern GPUs. More actualized information about NVidia and ATI implementations surfaces as unofficial or unconfirmed information on Internet forums [3][4]. Outside shader microarchitecture some recent works can be found. For example, T. Aila et al. proposed delay streams [5], as a way to improve the performance of immediate rendering GPU architectures with minor pipeline modifications and at the Graphics Hardware 2004 conference a simulation framework for GPU simulation, QSilver [7], was presented.

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 [6] 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 [21] and Falanx [22], 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 the 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 a middle-end PC GPU to a low-end embedded GPU. We have presented a low budget embedded GPU with a single unified shader that performs the triangle setup computations in the shader, saving area. We have evaluated the performance of the different configurations ranging from a baseline middle-end PC GPU architecture to the proposed embedded GPU architecture. We have also evaluated the cost of performing triangle setup in the unified shader unit for different graphic traces.

The proposed embedded architecture achieves with a single unified shader unit and ROP pipe, keeping with the low power, area and memory requeriments, a sustainable

rate of 20 frames per second with the selected Unreal Tournament trace at a resolution of 320x240.

References

[1] Erik Lindholm, et al. An User Programmable Vertex Engine. ACM SIGGRAPH 2001.

[2] WO02103638: Programmable Pixel Shading Architecture, December 27, 2002, NVIDIA CORP.

[3] Beyond3D Graphic Hardware and Technical Forums. http://www.beyond3d.com

[4] DIRECTXDEV mail list. http://discuss.microsoft.com/archives/directxdev.html

[5] T. Aila, V. Miettinen and P. Nordlund. Delay streams for graphics hardware. ACM Transactions on Graphics, 2003.

[6] T. Akenine-Möller and J. Ström Graphics for the masses: a hardware rasterization architecture for mobile phones. ACM Transaction on Graphics, 2003.

[7] J. W. Sheaffer, et al. A Flexible Simulation Framework for Graphics Architectures. Graphics Hardware 2004.

[8] Marc Olano, Trey Greer. Triangle Scan Conversion using 2D Homogeneous Coordinates. Graphics Hardware, 2000.

[9] Michael D. McCool, et al. Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. Proceedings Graphics Hardware 2001.

[10] Green, N. et al. Hierarchical Z-Buffer Visibility. Proceedings of SIGGRAPH 1993.

[11] S. Morein. ATI Radeon Hyper-z Technology. In Hot3D Proceedings - Graphics Hardware Workshop, 2000.

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

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

[14] Microsoft Meltdown 2003, DirectX Next Slides. http://www.microsoft.com/downloads/details.aspx?FamilyId=3319E8DA-6438-4F05-8B3D-B51083DC25E6&displaylang=en

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

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

[17] Volumetric Lighting II. Humus 3D demos: http://www.humus.ca/

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

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

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

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

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

[23] Victor Moya , Carlos Gonzalez, Jordi Roca, Agustin Fernandez, Roger Espasa. Shader Performance Analysis on a Modern GPU Architecture. IEEE Micro-38 2005.

[24] GPUBench. http://graphics.stanford.edu/projects/gpubench/

[25] Kayvon Fatahalian, Jeremy Sugerman, Pat Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. Graphics Hardware 2004.