A New Software Based GPU Framework

by

Evgeny Miretsky

A thesis submitted in conformity with the requirements for the degree of Masters of Applied Science

Computer Engineering
University of Toronto

© Copyright by Evgeny Miretsky 2013
A New Software Based GPU Framework

Evgeny Miretsky

Master of Applied Science

Computer Engineering
University of Toronto

2013

Abstract

A software based GPU design, where most of the 3D pipeline is executed in software on shaders, with minimal support from custom hardware blocks, provides three benefits, it: (1) simplifies the GPU design, (2) turns 3D graphics into a general purpose application, and (3) opens the door for applying compiler optimization to the whole 3D pipeline.

In this thesis we design a framework and a full software stack to support further research in the field. LLVM IR is used as a flexible shader IR, and all fixed-function hardware blocks are translated into it. A sort-middle, tile-based, architecture is used for the 3D pipeline and trace-file based methodology is applied to make the system more modular. Further, we implement a GPU model and use it to perform an architectural exploration of the proposed software based GPU system design space.
Acknowledgments

I would like to thank my supervisor Jianwen Zhu for guiding me through my thesis and providing valuable life advice, as well as being patient with my extracurricular career interests. I would also like to thank my parents Elena and Alex for supporting and encouraging me through the process.
Table of Contents

Acknowledgments............................................................................................................. iii

Table of Contents ............................................................................................................ iv

List of Tables .................................................................................................................. vii

List of Figures .................................................................................................................. viii

1 Introduction .................................................................................................................. 1
   1.1 Motivation............................................................................................................... 1
   1.2 Contributions ........................................................................................................ 2
   1.3 Outline ................................................................................................................... 4

2 Background ................................................................................................................... 5
   2.1 Low Level Virtual Machine (LLVM) .................................................................... 5
   2.2 3D rendering and 3D Pipeline ............................................................................. 6
   2.3 Gallium3D ............................................................................................................ 8
      2.3.1 Gallium3D High Level Design ................................................................... 8
      2.3.2 Pipe Driver Design ..................................................................................... 10

3 Related Work ............................................................................................................... 12
   3.1 Mobile GPU Design ........................................................................................... 12
      3.1.1 Immediate Mode Rendering (IMR) .............................................................. 12
      3.1.2 Tile Based Deferred Rendering (TBDR) ...................................................... 14
      3.1.3 Tradeoffs of IMR vs. TBDR ...................................................................... 16
   3.2 Software Based GPU Design .............................................................................. 17
      3.2.1 NVidia Cuda Based Renderer .................................................................... 17
      3.2.2 Intel Larrabee .............................................................................................. 19

4 Design Considerations ............................................................................................... 21
   4.1 Trace Based Design Methodology ...................................................................... 21
4.2 LLVM IR as Shader ISA ........................................................................................................22
4.3 Tile-based Architecture .........................................................................................................24
5 Driver Implementation ..............................................................................................................26
  5.1 LLVM Program Creation .......................................................................................................26
    5.1.1 Vertex Shader Program ...............................................................................................26
    5.1.2 Setup Program: ...........................................................................................................27
    5.1.3 Fragment Program .......................................................................................................27
  5.2 Driver Architecture .............................................................................................................27
    5.2.1 Draw Module ..............................................................................................................28
    5.2.2 Triangle setup: ...........................................................................................................30
6 GPU Model ..............................................................................................................................31
  6.1 Architecture .......................................................................................................................31
    6.1.1 Command Dispatcher .................................................................................................32
    6.1.2 Core ............................................................................................................................32
  6.2 GPU Model Implementation ...............................................................................................34
    6.2.1 LLVM Backend ...........................................................................................................34
    6.2.2 MVM Backend ...........................................................................................................34
7 Architectural Exploration ..........................................................................................................36
  7.1 Workload and Simulation Method .......................................................................................36
  7.2 Simulation Speed ...............................................................................................................37
  7.3 Number of Cores Effect on Performance and Load Balancing ........................................38
  7.4 Number of Threads per Core Effect on Performance and Core Utilization ......................40
  7.5 Texture Latency Effect on Performance ............................................................................42
  7.6 Tile Size Effect on Performance and Core Utilization .....................................................43
8 Conclusion ............................................................................................................................46
References ........................................................................................................................................47
List of Tables

Table 1: Benchmarks .......................................................................................................................................... 36

Table 2: Simulation Speed .................................................................................................................................... 38

Table 3: Statistics from varying the number of cores while keeping all other caribale constant at 64x64 tiles, 1 threads/core, 500 cycles texture latency and async states on .................................................. 39

Table 4: Statistics from varying the number of threads while keeping all other variables constant at 64x64 tiles, 4 cores, 500 cycles texture latency and async states on ......................................................... 41

Table 5: Statistics from varying the texture latency and number of threads while keeping all other variables constant at 64x64 tiles, 1 core, and async states on ................................................................................................. 43

Table 6: Statistics from varying the tile size while keeping all other variables constant at 1 core, 8 threads/core, 500 cycles texture latency and async states off ......................................................................................... 45
List of Figures

Figure 1: 3D Pipeline .................................................................................................................. 7

Figure 2: Gallium3D System .................................................................................................... 10

Figure 3: NVIDIA GForce series GPU Architecture ............................................................. 13

Figure 5: PowerVR5 GPU Architecture .................................................................................... 15

Figure 6: NVIDIA Software Based 3D Renderer ...................................................................... 18

Figure 7: High Level Trace-based Architecture ...................................................................... 21

Figure 8: Tile Based 3D Pipeline ............................................................................................... 24

Figure 9: Vertex Shader Program ............................................................................................. 25

Figure 10: Fragment Shader Program ......................................................................................... 26

Figure 11: Driver Architecture .................................................................................................. 28

Figure 12: Vertex Splitting Algorithm ....................................................................................... 29

Figure 13: Line Anti-aliasing ..................................................................................................... 29

Figure 14: Software GPU Model ............................................................................................... 31

Figure 15: Tiling Algorithm ....................................................................................................... 33

Figure 16: SOA to AOS Conversion Algorithm ....................................................................... 33

Figure 17: MVM Program Generation Flow ........................................................................... 35

Figure 18: The effect of increasing the number of cores on performance, as measured by the speedup compared to a one core configuration. All other variables are kept constant: 64x64 tiles, 1 thread/core, 500 cycle texture latency and async state on. .............................................. 40
Figure 19: The effect of increasing the number threads/core, as measured by the speedup compared to a one thread/core configuration. All other variables are kept constant: 64x64 tiles, 4 cores, 500 cycle texture latency and async state on.................................................................42

Figure 20: The speedup achieved by reducing the texture latency from 500 cycles to 300 cycles on performance for various number of threads/core. All other variables are kept constant: 64x64 tiles, 1 core and async state on......................................................................................44

Figure 21: The effect of increasing the tile size on performance, as measured by the speedup compared to an 8x8 tile configuration. All other variables are kept constant: 64x64 tiles, 1 thread/core, 500 cycle texture latency and async state on. .........................................................................................45
1 Introduction

1.1 Motivation

With the rapid development of smart phones, and tablets, there has been a widespread adoption of Graphic Processing Units (GPUs) in the embedded space. The hand-held market is extremely profitable and competitive, and is fueled by the high end-user demand for a continuously improved graphical experience. Unfortunately, mobile devices are powered by batteries with limited capacity, therefore enforcing strict requirements on the power consumption of mobile GPUs. It is also worth noting that the battery capacity is increasing at a modest pace of 5-10% per year, while the power consumption of other components is increasing with every generation [1]. A study of real smart phone users has shown that 44% of the power is consumed by the communication subsystems, leaving the GPU with a limited power budget [2]. With the ever increasing screen sizes and improving 3D performance, it is clear that mobile GPUs will soon be bound by power consumption. GPUs are approaching a “power wall”, and architecture innovation is required to circumvent this wall.

Most of the improvement in mobile 3D graphics over the last decade can be attributed to Moore’s law, as manufacturers relay on the ability to consistently increase the amount of logic and memory on a single chip. The innovation in the field has mostly focused on making incremental modifications to the traditional pipeline and instruction set architecture (ISA). Modern state of the art mobile GPUs are either a scaled down variation of the immediate mode rendering(IMR) architecture used by discrete GPUs [3], or a tile-based architecture [4][5]. In both cases the GPU has a number of programmable shader cores that execute vertex, fragment, geometry and tessellation shaders as well as fixed function units that performs tasks such as triangle setup, rasterization, raster operations, and scheduling.

Since NVIDIA introduced CUDA in 2007[6], programmable shader cores have also been exposed for general purpose programming (GPGPU). Furthermore, we are now seeing a paradigm shift towards heterogeneous computing [7], where software is written to run seamlessly on different hardware (CPU, GPU, etc.), while allowing each task to execute on the processor that will result in the best performance and/or energy efficiency. In such system, the
complexity of the GPU’s fixed function units proves to be an archaic burden that increases time-to-market and design costs.

In an ideal situation, most of the 3D graphic acceleration would be done on the shaders in software, with support from a minimal number of essential hardware blocks. The benefit of such software 3D pipeline is threefold. First, it would allow simplifying the GPU hardware by removing 3D graphics specific units. Second, it will turn 3D graphics into a general purpose application that can take advantage of emerging heterogeneous frameworks. Finally, it will open the door for exploring various compiler and ISA architecture techniques for reducing power consumption; for example, register file energy reduction [8], power gating [9], software micro-polygon rasterization [10], automatic shader bounding [11], and pre-fetching [12].

The first step towards building a new architecture is creating a robust framework that can be used for developing, testing, and simulating the design. Currently, several high quality open-source GPU simulators exist. Attila[13] is a cycle-level execution-driven GPU simulator that is used exclusively for 3D graphics workloads. GPGPU-Sim [14] provides a detailed simulation model of a contemporary GPU running CUDA and/or OpenCL workloads. Ocelot [15] is a dynamic just-in-time (JIT) compilation framework for GPUs and heterogeneous architectures that uses LLVM[16] to allow execution of CUDA kernels on a range of targets. However, all of the above simulators are designed for specific use-cases and workloads; hence lack the flexibility and tools to execute 3D graphic workloads in software on GPU shader processors, and/or heterogeneous systems.

The motivation for this thesis is to address the lack of specific tools for exploring software-based 3D graphics rendering on shader processors.

1.2 Contributions

In this thesis our mission is to build a framework that can be used for further exploration of software 3D rendering, as well as design of new shader ISAs. We leverage LLVM intermediate representation (IR) to create a complete tile-based 3D rendering pipeline that executes mostly in software with the assistance of a few simple hardware blocks. For such framework to be practical in real life, it has to be (a) easy to use and debug, (b) flexible enough to allow explorations of the full design space, and (c) fast enough to allow rapid prototyping.
Today’s GPU simulators can be roughly categorized into 2 groups: slow cycle-accurate GPU simulators, and flexible GPGPU simulators. For example, Attila [13] provides a cycle-accurate GPU simulator that is very slow and has built-in ‘hard’ pipeline stages that are not flexible enough to allow executing the entire pipeline in software. Ocelot [15], on the other hand, allows fast execution of shader code using JIT compilation, but lacks support for 3D graphics workloads.

In this thesis we combine the best of both groups to build a framework that meets all the above criteria; more specifically, the four main contributions of this thesis are:

1. We describe an implementation of a framework that can be used for further exploration of software 3D rendering. The system implements a tiles-based 3D pipeline, where the scene is decomposed into tiles that can be rendered independently in parallel. Furthermore, we use LLVM IR as an intermediate shader representation allowing the framework to be easily retargeted for new ISA architectures. We further demonstrate the completeness of our framework by showing that developers can develop, debug, and enhance components separately by using a trace-based methodology.

2. We provide a flexible model of tile-based renderer with a full memory hierarchy implementation. The model supports two separate ISAs: LLVM IR, and custom IR. We demonstrated the completeness of the whole framework by playing back non-trivial games written using OpenGL [17] standard.

3. We show that by translating most of the 3D pipeline into shader code, and then compiling the shader code into native instructions we can achieve an order of magnitude faster simulation time.

4. We perform detailed architectural exploration of the design space of the proposed architecture by varying the design parameters of the system. We further analyze the results to draw conclusions about the scalability of the system, and to make recommendation on how to improve performance.
1.3 Outline

In order to fully understand the design of the software based GPU system architecture, Chapter 2 provides background on the 3D graphics pipeline as well as some of the 3rd party projects used to build our system. Chapter 3 briefly describes related work that has been done in the field. Chapter 4 describes some of the design decisions that shaped the system. Chapter 5 describes the driver implementation and Chapter 6 describes the GPU model implementation. Chapter 7 describes the results of a series of experiments we conducted, showing that our system can be successfully used for architecture exploration of the software based GPU design space. Finally, chapter 8 concludes our work.
2 Background

In this section we go over general 3D graphics concepts and give a high level overview of the 3D graphics pipeline. We also briefly describe the two software interfaces that are used to build our system: (i) Gallium3D graphics driver, and (ii) the LLVM compiler framework and intermediate representation (IR) that is used to represent shaders in our system.

2.1 Low Level Virtual Machine (LLVM)

LLVM [18] is a compiler infrastructure designed for compile-time, link-time and run-time optimisations of programs written in an arbitrary programming language. It is a set of reusable libraries with well-defined interfaces that perform various compiler related tasks, and are held together by the LLVM intermediate representation (IR). In a typical LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR. This IR is optionally fed through a series of analysis and optimization passes which improve the code, and then is sent to a code generator to produce native machine code. Furthermore, as its name suggests, LLVM can be used as a virtual machine to execute programs represented in LLVM IR. Such programs can be executed by LLVM using interpretation or JIT-compilation. LLVM frontends for C, C++, Objective-C, and FORTRAN are available. LLVM supports many target architectures through different back-ends, including X86, PowerPC, ARM, Thumb, SPARC, Alpha, PTX and CellSPU.

LLVM IR is the most important component of the LLVM framework. It was designed to provide high-level information about programs that is needed to support sophisticated analyses and transformations, while being low-level enough to represent arbitrary programs and to permit extensive optimization. The LLVM IR can be represented in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. The LLVM instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipelines, and low-level calling conventions. LLVM provides an infinite set of typed virtual registers which can hold values of primitive types (boolean, integer, floating point, and pointer). The virtual registers are in Static Single Assignment (SSA) form [19]. LLVM is a load/store architecture: programs transfer values
between registers and memory solely via load and store operations using typed pointers. Vectors are supported as a native type and binary and bitwise vector instructions are provided as well as instruction to insert and extract vector elements.

Furthermore, in order to support target-architecture specific functionality, LLVM IR provides intrinsic functions. Without specific handling, unknown intrinsics are handled like external function calls during all LLVM passes. Intrinsics are well suited for target dependent instructions like texture look-ups or mathematical functions which are implemented in hardware on some targets.

This thesis utilizes LLVM in several ways. First LLVM IR is used as the intermediate shader representation. Second, the LLVM JIT module is used to execute LLVM IR in the LLVM GPU model implementation. Third, a code generator backend is implemented to compile LLVM IR intro the target architecture ISA - MVM IR in this case.

2.2 3D rendering and 3D Pipeline

A 3D rendering algorithm takes as input a stream of primitives (triangles, lines, etc.) that define the geometry of a scene and processes them to produce a 2D image. 3D pipeline refers to the sequence of steps required to implement the rendering algorithm. Figure 1 shows a simple implementation of a modern 3D pipeline. The primitives are processed in a series of steps, where each step forwards the results to the next one. Some of the stages are traditionally implemented using dedicated logic, while others are shader programs that get executed on the shader processor.

The pipeline consists of the following stages:

**Streamer:** The streamer takes as input a stream of primitives (usually triangles), breaks them down into vertices and feeds them to the vertex shader. Usually the streamer also allows reusing vertices that are shared between primitives, and have already been shaded by storing the shaded vertices in a small cache.

**Vertex Shader:** Vertex shaders operate on individual vertices; their purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen. Vertex shaders can manipulate properties such as position, color and texture coordinates, but
cannot create new vertices. Vertex shaders enable powerful control over the details of position, movement, lighting, and color.

**Primitive assembly:** Is responsible for assembling the shaded vertices back into primitives.

**Clipper:** Is responsible for performing trivial triangle rejection test to eliminate triangles that are outside of the view frustum.

**Geometry Shader:** Geometry shaders are optional; they take as input a whole primitive, and then emit zero or more new primitives. It is usually used for special effects such as: displacement mapping, cube map generation, extrusion of shadow volumes, etc[20].

**Triangle Setup and Rasterizer:** The triangles get converted into fragments, where a fragment represents a part of the triangle that covers a single pixel. In modern GPUs this stage is usually broken into several steps. First the edge equations for the triangle are computed in the triangle setup stage. Then each fragment is checked against the line equations to test if it is inside the triangle. Finally, the value of each vertex attribute (color, etc.) is interpolated for each fragment. In a trivial implementation, the value of the attribute is calculated as the weighted average of the attribute’s value at the three vertices. However, all modern GPUs support perspective correct interpolation to account for the triangle’s ordination.

**Fragment Shader:** Fragment shaders compute the final color value of each fragment. They can alter the depth of the fragment (for Z-buffering), or output more than one color if multiple render targets are active. Fragment shaders make a heavy use of textures to produce special effects such as bump mapping, fog, lighting, etc.
Post Fragment Operations: Is a sequence of stages that are responsible for determining the final color of on-screen pixels based on the fragments produced by the fragment shader. First depth test is performed to determine if the fragment is hidden by already rendered fragments. Then blending is performed to blend the color of the fragment with the color of the already rendered pixel.

2.3 Gallium3D

Gallium3D is an open source library for 3D graphics device drivers that was originally developed by Tungsten Graphics (now part of VMware) and aims to replace the old Mesa 3D driver. It has been part of the Mesa open source driver since 2009 and is being used by open source graphics drivers from NVidia, ATI and IBM. While the old Mesa driver is built on top of complex legacy code, Gallium3D presents a new architecture that is designed from the ground up to support modern GPUs. This architecture introduces 4 main innovations: modular design, abstraction of modern GPU architecture, JIT compilation of shaders using LLVM and translation of hard 3D pipeline stages into shaders:

2.3.1 Gallium3D High Level Design

A typical graphics system consists of several components:

- GPU: Responsible for executing the actual functionality of 3D rendering. In most cases it is a HW GPU, however it could also be a software GPU emulator or a reference software implementation of the 3D graphics pipeline.
- Windowing System: Responsible for 2D operations such as drawing images and sharing the desktop between multiple application as well as controlling the mouse and keyboard. In a typical Linux environment this is implemented by the X windowing system.
- Application API: An API used by developers to create 3D graphics scenes. Typically it is implemented by DirectX on windows, OpenGL on Linux and OpenGL-ES on embedded systems. However, theoretically, it could also be any API that takes advantage of GPU hardware capabilities, such as OpenVG for vector graphics of OpenCL for GPGPU programming.
- Operating System/Kernel driver: When a GPU is present, the driver is responsible for facilitating communication between the user-level driver and the hardware. Specifically,
it manages the framebuffer, GPU memory allocation, command queues, and synchronization

- **Graphics drivers**: Responsible for communicating between the mentioned above components. In particular, it translates the API calls received from the application into a set of commands for the hardware (or software GPU), and then calls the windowing system to display the results. The driver is usually very large and complex (AMD radeon ~ 90k lines of code).

The old Mesa driver implementation supports mainly OpenGL as an application API, and X-server as a windowing system. There is very little code reuse as each GPU model has its own driver. Not only that bringing up a new GPU model requires rewriting a big chunk of the driver, supporting a new windowing system or application API would require making modifications to all the drivers for all the different GPUs.

Gallium3D was designed in part to provide a more modular design that will facilitate code reuse and faster development time. The architecture of Gallium3D is shown in Figure 2.

Gallium3D consists of 4 components:

- **State Tracker**: Interfaces between the application API and Gallium3D pipe driver. There are currently state trackers for OpenGL, DirectX, OpenGL-ES, and OpenVG as well as work being done on an OpenCL state tracker
- **Pipe Driver**: The core of the Gallium implementation that is responsible for GPU specific functionality. There is a pipe drive for each supported GPU, as well as a software pipe driver, and a JIT LLVM pipe driver.
- **WinSys**: Responsible for the communication with the windowing system and OS in order to create renderable buffers, synchronize with other applications, and send commands to the hardware. While in theory there should be only one WinSys per windowing system, in practice there is one implementation per kernel driver module as kernel drivers differ slightly in the interface they expose for different GPU generations.
• Auxiliary modules: A set of modules that provide functionality that is shared across different pipe driver. It includes functions that support: vertex transformation, TGSI (Gallium3D’s internal shader IR) construction, state caching and assembly code generation.

Supporting a new application API on multiple GPUs is as easy as creating a new state tracker. Bringing up a new hardware is also easy as it is independent from the API, and requires testing only the pipe driver.

2.3.2 Pipe Driver Design

The most important component of Gallium3D is the pipe driver. It is the only component that has to be modified in order to bring up a new hardware. From this point onwards we will ignore the state tracker and winsys, and focus on the details of the pipe driver interface and design.

At the core of the Gallium3D design there are 3 main interfaces:

- pipe_screen: An abstraction of a GPU. It is responsible for querying GPU capabilities, creating resources, fencing and other miscellaneous tasks.
- pipe_context: An abstraction of the graphics pipeline. Essentially a collection of functions to create and bind states that represent the settings of different 3D pipeline stages: Shaders (vertex, fragment and geometry), rasterizer, per-fragment operations (blending, depth/stencil test, etc), texture sampler. Also, functions for drawing vertices are provided.
- TGSI (Tungsten Graphics Shader Infrastructure): The internal shader IR used by Gallium3D. All TGSI instructions operate on arbitrary-precision floating-point four-component vectors. An opcode may have up to one destination register and between zero and three source registers. Some instructions permit re-interpretation of vector components as integers, while other permit using registers as two-component vectors with double precision. When an instruction has a scalar result, the result is replicated to all 4 destination components.

Every pipe driver must implement the pipe_screem and pipe_context interfaces as well as translate TGSI into its native shader assembly language. Since the above interface is very close to modern hardware GPU architecture, a typical pipe driver is very simple. The driver’s main job is to translate the state change calls into a sequence of commands to updates the same state on the hardware and to manage the command batch buffer. For comparison, the Gallium3D Intel i965 driver contains only 28k LoC, most of which are responsible for: shader compilation, debugging, or simple maintenance tasks.
3 Related Work

In this section we explore the state of the art technology in mobile GPU design, and discuss recent attempts at building software based 3D renderer.

3.1 Mobile GPU Design

A graphics processing unit (GPU) is a dedicated parallel processor optimized for accelerating graphical computations. The GPU is designed specifically to perform the many floating-point calculations essential for 3D graphics rendering. Modern desktop GPU processors are massively parallel, fully programmable and can deliver as much as 4.58 teraflops single precision floating point performance[21].

Mobile devices are not permanently plugged in, so power efficiency is a key consideration in the design and performance of mobile GPUs. Early days mobile GPUs were simple single core fixed-point processors that were capable of delivering only a fraction of the performance of desktop GPUs. In 2004, mobile phones were running at a 320x240 pixels resolution, and were capable of rendering only trivial graphics scenes [22]. Nowadays, mobile GPUs’ power displays as large as 1136x640 pixels, render complicated games and user interfaces[23], and support GPGPU computing through OpenCL[24][25]. As discussed the next sections 3.1.1 and 3.1.2, commercial mobile GPUs architectures can be divided into two major groups: Immediate Mode Rendering (IMR), and Tile Based Deferred Rendering (TBDR).

3.1.1 Immediate Mode Rendering (IMR)

A traditional Immediate Mode Rendering (IMR) architecture is given its name because each submitted object travels through the entire pipeline immediately. Figure 3 shows the architecture of a NVIDIA Force 6 series GPU [26]. It was originally launched in 2004 as a desktop GPU architecture, and now a scaled down version is used for the NVIDIA’s Tegra system on chip.

A command stream is written by the CPU, which initializes and modifies states, sends rendering commands, and references the texture and vertex data. Commands are parsed, and a vertex fetch unit is used to read the vertices referenced by the rendering commands. The commands, vertices, and state changes flow downstream, where they are used by subsequent pipeline stages.
The vertex processors (VS) consists of a 32bit floating point vector processor, a 32bit floating point scalar process, branch unit, texture unit, primitive assembly unit and a viewport transformation unit. The vertex shader program is executed on each vertex in the object, performing transformations, skinning, and any other per-vertex operations. There is also a vertex cache that stores vertex data both before and after the vertex processor, reducing fetch and computation requirements. This means that if a vertex index occurs twice in a draw call (for example, in a triangle strip), the entire vertex program doesn’t have to be re-run for the second instance of the vertex—the cached result is used instead.

![NVIDIA GForce series GPU Architecture](image-url)

**Figure 3:** NVIDIA GForce series GPU Architecture
Viewport transformation is then applied, and vertices are grouped into primitives, such as: points, lines, or triangles. The Cull, Clip and Setup blocks all operate on individual primitive. First, removing primitives that are not visible, then clipping primitives that intersect the view frustum, and, finally, setting up the edge and plane equations for later use in the rasterization stage. The rasterizer unit calculates which fragments are covered by each primitive, and can optionally use the z-cull block to perform an early depth test - quickly discarding fragments that are hidden by previously rendered objects.

The texture and fragment processing unit (FS) operate on squares of four pixels quads at a time, allowing for direct computation of derivatives for calculating texture level of detail. The fragment processor works on groups of hundreds of pixels at a time in single-instruction-multiple-data (SIMD) fashion – the same instruction is performed on all the fragment quads, before moving on to the next instruction. As a result, even if the pipeline is stalled for one fragment quad due to texture fetch latency, the processor continues working on the other fragments, resulting in no performance lose. After fragment-processing fragments are sent to the z-compare and blend units, which perform depth testing, stencil operations, alpha blending, and the final color write to the target surface.

The number of actual vertex processors, fragment processors, and blending units is flexible, and can be scaled in order to service different performance-price-power points.

### 3.1.2 Tile Based Deferred Rendering (TBDR)

In tile-based deferred rendering hardware, rasterization is performed on a per tile basis instead of rasterizing the entire scene as a traditional IMR would. To facilitate this, rendering is performed in two steps: first all geometry is processed, transformed into screen space, and divided into tiles; then each tile is rendered independently from the other tiles. This enables the hardware to preload the color, depth and stencil buffers for each tile into a local memory. Read-modify-write operations are performed on fast local memory, and traffic to system memory is reduced.

Deferring rendering primitives until the whole scene has been transformed into screen space enable more effective hidden surface removal (HSR). In a traditional IMR rendering hardware, fragments are rendered in primitive submission order, resulting in fragments that are further from the camera being drawn, and then overdrawn by fragments that hide them. This results in redundant fragment processing, and expensive memory access operations. Ensuring that only visible fragments are processed not only improves performance, but also reduces power
consumption. In typical TBDR hardware, a HSR block processes all primitives for a given tile, and ensures that only visible fragments are passed to the next state.

Figure 4 shows a TDBR architecture based on the design of Imagination PowerVR5 [27]. Most of the components are similar to the IMR GPU described in the previous section. The difference is that after primitives are processed, the tiling block tests each primitive to determine which tiles it covers and writes it into two data structures in memory: a transformed vertex data buffer that stores processed primitives, and a per tile linked list that references the primitives in vertex data buffer.

Figure 4: PowerVR5 GPU Architecture

The visibility test block is responsible for per-tile HSR, ensuring that the only fragments processed by the fragment processor are those that will affect the rendered image. To do this, all of the triangles referenced in the tile’s primitive list are processed one-by-one, calculating the triangle equation and, if depth testing is enabled, projecting a ray at each position in each triangle to retrieve accurate depth information for each fragment that the primitive covers within the tile. The calculated depth information is then compared with the values in the tile’s on-chip depth buffer to determine if the fragments are visible. A Tag Buffer is used to track all visible fragments and the primitives that they belong to. When all of the primitives that the tile’s
Primitive List references have been processed, fragments are submitted to the fragment processor in groups of fragments that belong to the same primitive, rather than submitting visible fragments on a per scan line basis. Doing so allows the hardware to improve the efficiency of cache access and fragment processing.

### 3.1.3 Tradeoffs of IMR vs. TBDR

When comparing immediate mode rendering to tile-based deferred rendering, it is important to consider the tradeoffs between the two architectures.

TBDR, when compared to IMR offers the following benefits:

- Fast access to color, depth and stencil buffers on the on-chip memory. In addition to saving time it takes to transfer data over the system’s memory bus, accessing data on a local memory significantly decreases power consumption.

- Obscured fragments are removed early, and never get processed. While IMR architectures utilize Early-Z techniques to perform depth tests early in the graphics pipeline to reduce the amount of overdraw, applications can only fully benefit from this optimization if geometry is always submitted to the hardware in front to back order. Less rendering work results in reduced shader utilization, memory traffic and power consumption.

- Easily scalable design. Since each pipe/core work on separate tiles, it is possible to just add more renderer pipes as each pipe works on its own tile.

And the following disadvantages:

- A local memory buffer is needed for sorting the triangles in each tile, resulting in more on-chip memory.

- Triangles covering several tiles need to be processed several times, ones for each covered tile.

- Unpredictable performance as performance depends on the structure of the scene as opposed to only the number of rendered triangles.
Considering the significant power benefits offered by TBDR, most mobile GPU vendors seem to prefer some variation of a tile based architecture.

### 3.2 Software Based GPU Design

Modern GPUs are increasingly programmable in order to support advanced graphics algorithms and other parallel applications. However, full programmability of the graphics pipeline is restricted by fixed function blocks such as rasterization and post-shader blending. Constructing a purely software based graphics pipeline will open the opportunity to augment it with various extensions that are impossible or infeasible to fit in the hardware pipeline (without hardware modifications, that is).

In a software based GPU all stages of the 3D pipeline are translated into a software shader programs and are executed on either the shader processor, or a custom processor. However, most real world implementations include a few hardware blocks to accelerate software rendering. Two such implementations are presented bellow

#### 3.2.1 NVidia Cuda Based Renderer

NVIDIA built an experimental software based GPU that executes the entire pipeline using CUDA[28]. CUDA (Compute Unified Device Architecture) is a parallel computing on GPUs platform developed by NVIDIA. It allows general purpose applications to take advantage of the shader processors and memory of a GPU. The goal of the project was to explore the tradeoffs of such architecture, and to identify potential performance hot spots in the software pipeline. In CUDA programming, a SP (steam processor) represents a GPU shader core with its own local fast shared memory. The programmer creates “threads”, which are the smallest units of processing that, can be scheduled to run on SPs in parallel. The threads are grouped into ‘warps’ of 32 threads, which in turn are grouped into blocks of up to 512 threads (on GeForce GTX 480). Threads within the same block can synchronize with each other, and can access the same local shared memory. Communicating between threads in separate blocks is expensive and can result in significant performance loss. For a more comprehensive description of the execution model, please refer to [29].
Figure 5 illustrates the high level design of the software based 3D pipeline. The design implements a tile-based deferred rendering 3D pipeline that consists of four stages: triangle setup, bin rasterizer, coarse rasterizer, and fine rasterizer. Each stage is executed as a separate CUDA kernel launch. Triangle setup performs culling, clipping, snapping, and calculation of plane equations. Bin rasterizer and coarse rasterizer generate, for each tile, a queue that stores the triangle IDs that overlap the tile. The last stage, fine rasterizer, processes each framebuffer tile, computes exact coverage for the overlapping triangles, executes the shader, and performs ROP.

The pipeline was optimized to provide optimal parallelism in each stage. Triangle setup assigns a single triangle to each thread, and hence is trivial to scale. Fine rasterizer launches 20 warps per block, and assigns a tile to each warp. 32-bit RGBA color and 32-bit per pixel depth tiles are kept in shared memory, amounting to 10 KB with 8x8 pixel tiles and 20 warps per core. The rasterizer stage is responsible for splitting the triangles into bins, however with 8x8 pixel bins; even a 512x512 pixel scene will have 4096 tiles. In order to be able to keep all the bins for a single block in shared memory, this part of the pipeline is divided into two stages, bin rasterizer and coarse rasterizer. First triangles are rasterized into bins that contain 16x16 tiles (128x128 pixels), and then each bin is processed to produce per-tile queues.

The performance was evaluated on a GeForce GTX 480 with 1.5 GB of RAM, installed in a PC with 2.80 GHz Intel Core i7 CPU and 12 GB of RAM. It was found that the software pipeline is generally 1.5–8x slower than the hardware. The authors hypnotize that most of the performance loss was due to attribute interpolation, pixel coverage mask calculations, and raster operations. Furthermore, increasing the performance of shared memory atomics would help in almost all of
the pipeline stages, and provide perhaps 5–10% speedup. Adding a custom instruction to find the $n$th set bit in a coverage mask would increase performance by 2%-4%.

### 3.2.2 Intel Larrabee

Larrabee\cite{30} is a CPU-like x86-based GPU architecture that was designed by Intel. Its planned release in 2010 was delayed due to technique difficulties, and the project was eventually canceled. Larrabee is designed around multiple instantiations of an in-order, 4 way hyper threading, CPU core that is augmented with a wide vector processor (VPU), L1 cache, and a 256KB local subset of a L2 cache. Some operations that GPUs traditionally perform with fixed function logic, such as rasterization and post-shader blending, are performed entirely in software in Larrabee. Like GPUs, Larrabee uses fixed function logic for texture filtering.

Larrabee uses 2 phases, tile-based deferred rendering architecture. In the front-end, each set of triangles that share a common rendering state (PrimSet) is given a sequence ID to identify where in the rendering stream it was submitted. The PrimSet is then assigned to a single core, which performs vertex shading, tessellation, geometry shading, culling and clipping to produce triangles (or other primitives). The core then rasterizes each triangle to determine which tiles it touches and which samples it covers within each of those tiles. The result is a series of X, Y coordinates and sample coverage masks for each triangle. This data is stored in the bins along with indices that reference the vertex data.

Once all front-end processing for the RTset has finished and every triangle has been added to the bin for each tile that it touched, back-end processing is performed. Here, each tile is assigned to a single core, which, with the assistance of texture co-processors, shades each triangle from the associated bin. The back-end then performs depth, stencil and blending operations.

Unlike a regular GPU, Larrabee performs rasterization in software. The used algorithm is a highly optimized version of the recursive descent algorithm \cite{31}. The basic idea is to convert clipped triangles to screen space, and then compute a half-plane equation for each triangle edge \cite{32}. This helps determine if a rectangular block is entirely inside the triangle, entirely outside the triangle, or partially covered by the triangle. In the latter case, the algorithm subdivides the block recursively until it is reduced to an individual pixel or sample position.
The back-end code starts by pre-fetching the color and depth buffer tiles into L2 cache. Then a setup thread reads primitives for the tile. Next, the setup thread interpolates per-vertex parameters to find their values at each sample. Finally, the setup thread issues pixels to the three work threads in groups of 16. The three work threads perform all remaining pixel processing, including pre-shader early Z tests, pixel shading, regular late-Z tests, and post-shader blending. In order to hide texture access latency, many batched of 16 pixels are processed on a single hardware thread. After every texture access, the thread switches to processing a different batch.
4 Design Considerations

Our system is designed to provide a flexible framework for architectural exploration of software based GPU rendering. The first requirement of such system is that it must integrate well with existing 3D graphics applications. The second requirement is for it to be easy to develop and debug. When working on any large system, it is of the utmost importance to be able to develop and test components independently of each other. To this end, a modular design with well defined interfaces was created. Furthermore, the design is well integrated with Gallium3D, and hence can be easily run on any Linux system. Figure 6 presents the high level design of our system.

![Figure 6: High Level Trace-based Architecture](image)

4.1 Trace Based Design Methodology

Traditional 3D graphics systems are broken into three components: application, driver, and GPU. Separating the three components provides the benefit of separate and well defined functionalities. The application is responsible for creating the 3D graphics scene, the GPU is responsible for executing the graphics workload, and the driver is responsible for communicating between the two. We further separate these components by introducing trace files between them.

One major source of agony when developing 3D drivers is the nondeterministic nature of 3D graphics applications. Playing the same game twice will result in completely different call sequences. This in turn makes bugs hard to reproduce, and makes comparing different architectures less accurate. In our design, each 3D application is first run using the Gallium3D trace driver. The trace driver processes the OpenGL commands on the native hardware, while dumping all Gallium3D pipe driver interface calls and buffer content to a trace file. Our driver never communicates directly with the actual application. Instead a python module reads the trace file and calls the driver one pipe interface call at a time, as if the calls were issued by the application.
The process of communication between the driver and the GPU can differ from one system to another as communication usually involves the OS kernel in the case of a hardware GPU, or inter-process communication in the case of a software graphics accelerator. However, details of the communication process are nonessential to the overall architecture as the only important information is the actual data and commands issued by the driver. To abstract this away, a trace file is introduced between the drive and the GPU. The driver writes all the commands and buffer data to one trace file, and the shader IR representation to another trace file. Similarly to the driver, there is a python module that reads the trace files and passes the commands to the GPU one-by-one.

4.2 LLVM IR as Shader ISA

Our system is designed to explore the potential merit of a software based GPU - a design where all the stages of the 3D pipeline are executed in shader/software. In order to make such system as effective as possible, it is necessary to pay special attention to two aspects: the implementation of hardware 3D pipeline blocks in shader code, and the execution of the shader code on the GPU. Hence, the shader intermediate representation (IR) and compiler are vital to this design.

LLVM (Low Level Virtual Machine) IR was selected as the shader IR for our design. LLVM is an open source compiling, optimization, and code generation framework. LLVM provides a SSA like intermediate representation language (IR) that supports a large set of common optimizations. Code generation for many CPU targets is supported out of the box (SSE, AltVec, etc.), and a generic framework is provided to quickly develop new LLVM code generators for new target instruction set architectures (ISA).

The advantage of using LLVM IR is twofold. First, using LLVM allows easily retargeting the shader for a new ISA. LLVM is very well-documented, especially when compared to open source 3D drivers like mesa, and provided a detailed manual on how to write a new code generation backend. Second, LLVM comes with a mature optimization framework that supports just-in-time (JIT) compilation. More specifically, using LLVM IR provides the following benefits.

For software GPUs:
- A mature optimizations framework.
- Ability to execute code, and take advantage of machine specific optimizations on most common CPU architectures.

For hardware GPUs:

- Support for common optimization passes that can be applied to GPU shaders.
- A well documented methodology for retargeting the shader code to new hardware ISA by creating a new LLVM backend.
- Ability to execute some code both on the GPU and the CPU.
- Access to work done by a large open source community such as the PTX code generator. [13] [7]

Gallium3D stores shaders in Tungsten Graphics Shader Infrastructure (TGSI) format. It is the driver’s responsibility to translate the vertex and fragment shaders provided by Gallium3D (through the pipe interface) into LLVM IR. TGSI was specifically designed to support graphic shaders, and therefore inherently supports vector instructions (all instructions operate on 4-component vectors) as well as custom opcodes such as: lighting transformation, texture fetching, and complex math operations (cos, sin, etc.). When translating to LLVM IR, all TGSI instructions that have one-to-one mapping to LLVM IR instructions get translates into 4-element vector instructions. Graphics specific instructions can be treated in one of two ways:

- Get lowered into simple memory access and arithmetic instructions. For example, texture fetch is a single TGSI instruction that calculates the texture derivative, LOD level, and textual address, fetches the textual value, and then performs filtering. When it gets translated to LLVM IR, it is turned into a long sequence of simple arithmetic and memory load operations.
- Emitted as intrinsic instructions that are later implemented by target ISA. For example, the aforementioned texture instruction is emitted as an intrinsic call that is implemented by the hardware calling the texture unit.

The resulting program operates on inputs and outputs that are laid out in a SOA (structure-of-arrays) format, while OpenGL usually arranges data in an AOS (array-of-structures) format, thus
requiring the driver to perform a conversion.

As mentioned above, we attempt to translate most fixed function GPU stages into shader code. More specifically, triangle setup, attribute interpolation and post fragment shader operation (depth/stencil test, alpha test, blending) are translated into shaders. The only remaining fixed function unit is a small triangle rasterization unit. Details about how each fixed function unit gets implemented in shader are provided later in the thesis.

4.3 Tile-based Architecture

The key for achieving high performance for any parallel rendering algorithm is to divide the rendering task into many subtasks that can be load balanced and executed in parallel with very few synchronization points. To this end, a sort-middle software renderer that uses binning for load balancing was designed.

Figure 7 shows the high level design of the 3D pipeline implementation.

The surface being rendered is split into tiles of pixels. Each tile has a bin that will be filled with the triangles that intersect that tile. The terms tile and bin are sometimes used interchangeably. The distinction in this thesis is that a tile is the actual pixel data, while the bin is the set of triangles that affect that tile.

There are two phases to the processing. The driver performs vertex shading, tessellation, geometry shading, culling and clipping to produce triangles (or other primitives). Then an intersection test is performed on each triangle in order to determine which tiles it touches and
which samples it covers within each of those tiles. The result is a series of X, Y coordinates for each triangle. This data is stored in the bins along with the attributes values and setup planes.

The tile size is set to a default of 64x64 pixels, but can be easily customized, and would ideally depend on the memory architecture of the GPU. The main impact of using smaller tiles is that more triangles are likely to intersect multiple bins, and hence end up being processed twice. Using larger tiles, on the other hand, can result in a poor load balance as there is a higher change that some tiles will contain significantly more triangles than others.

The GPU assigns each tile to a single core. First, each triangle is rasterized, and the resulting fragments are passed to the fragment program. The fragment program performs attribute interpolation, and fragment processing as well as depth and stencil tests and blending.

Unlike some other tile-based rendering methods, there is no attempt at perfect occlusion culling before shading, reordering of shading, or any other non-standard rendering methods. Rendering for a single tile is performed in the order in which the commands are submitted. Using a conventional rendering pipeline within each tile avoids surprises in either functionality or performance.

![Figure 8: Vertex Shader Program](image-url)
5 Driver Implementation

The driver is responsible for several tasks. First, it executes the vertex shading and triangle setup stages of the 3D pipeline and divides the processed triangles into bins. Second, it sends state update and bin rendering commands to the GPU. Third, it is responsible for creating the vertex, triangle setup and fragment LLVM programs. This section provides a detailed description of the functionality and implementation of each component.

5.1 LLVM Program Creation

The driver is responsible for creating 4 LLVM programs: vertex shader, geometry shader, setup, and fragment. The first 3 are executed by the driver as part of vertex transformation and setup, while the latter gets executed by the GPU. The details of each of those programs are described below.

5.1.1 Vertex Shader Program

The vertex LLVM IR program gets executed ones for each vertex/element array. As illustrated in figure 8, the array is traversed in stamps of 4 vertices at a time. For each stamp, first the vertex attributes are fetched from the vertex buffers and converted from AOS to SOA format. Then the translated vertex shader code is run followed by code that generates the triangle clip mask by comparing vertices against the provided clipping planes. Finally, the results are converted back into AOS format and stored.

There are two variants of the vertex program - one for element arrays, and one for vertex arrays. The latter performance exactly the same operations as the former.

Figure 9: Fragment Shader Program
except that it iterates the vertices in steps of size one, while the former iterates the vertices using the index values from the element array.

5.1.2 Setup Program:

The setup program gets executed ones for every triangle and is responsible for setting up the coefficients that will later be used for attribute interpolation. Specifically, for each attribute, its value at the origin is computed together with the value derivative in the x and y direction.

5.1.3 Fragment Program

As illustrated in figure 9, the fragment program gets executed ones for every 4x4 fragment tile and is responsible for coordinate interpolation, fragment shading and post fragment operations such as blending, depth test, and alpha test. While the fragment shader part gets translated directly from the TGSI fragment shader, the rest of the stages are writing using LLVM IR, and are optimized to utilize vector instructions as much as possible.

A big source of complexity here is that different stages are better suited to run with different precision and data type. For example, the fragment shader is done in floats, while blending is better done in the native color buffer format (typically 8-bit unsigned integer). A typical SIMD register in most common architectures is 128 bit wide, and therefore can fit either 4 floats, or 16 8-bit integers. In order to take advantage of the entire width of a SIMD register, the tile is divided into 4 2x2 quads and the float part of the program is executed on individual quads in a loop. Then the resulting float color values of the 16 fragments get converted into 16 8-bit unsigned integers and blending is performed.

When the program starts, the interpolation coefficients are read from the input vertex data. For each attribute, the attribute’s value in the corner of each of the 4 quads is calculated. Then, for each quad, the attribute values are calculated for each of the 4 fragments, followed by the typical early depth test, fragment shading, alpha test and late depth test pipeline. Finally, blending is performed, and the color values get written into the color buffer in SOA format.

5.2 Driver Architecture

This section describes the architecture of the driver as shown in Figure 10.
A pluggable software vertex shading and rasterization module that is provided by the Gallium3D auxiliary library, and is used by pipe drivers for hardware that lacks vertex transformation capabilities. The Draw module has 4 stages.

5.2.1.1 Vertex Pre-fetch

The vertex shader can operate only on a limited number of vertices at a time. This limit is usually dictated by the number of vertices the driver can send to the GPU in one batch. If the vertex stream is larger than what that GPU can handle, the stream is split into smaller chunks.
The complexity here stems from the different primitive types that have to be handled and from the need to optimize the resulting index and vertex buffers. Figure 11 shows a stream with 10 vertices and 11 indices that is split into 2 chunks of 5 vertices. First, a new vertex buffer is created. Then the index buffer gets traversed one-by-one and if the corresponding vertex is not already in the new vertex buffer, it gets added to it. The output index buffer gets updated with the position of the vertex in the new vertex buffer. When the vertex buffer exceeds the maximum chunk size, a new pair of vertex and index buffers is created.

5.2.1.2 Vertex Transformation:

This stage is responsible for calculating the new position of each vertex and for optionally producing new vertices. First, all the vertices are fetched and converted into the format supported by the GPU. Then the LLVM vertex program is executed on all the vertices. Finally, if geometry shader is enabled, the vertices are assembled into simple primitives (point, line, triangle), and the LLVM geometry program is executed on each primitive.
5.2.1.3 Primitive Pipeline

A sequence of small stages that operate on simple primitives and perform various rasterization related operations such as clipping, anti-aliasing, stippling, etc. Some of the stages are performed using software modules; however, others get JIT compiled into the fragment shader. One such example is the line anti-aliasing stage. Line aliasing is the jaggedness that appears because the ideal line is approximated by a series of pixels that must lie on the pixel grid, while, in reality, as shown in figure 12, a one-pixel wide line covers multiple pixels. When performing anti-aliasing, one calculates a coverage value for each fragment based on the fraction of the pixel square on the screen that it would cover and sets the alpha value accordingly. The draw pipeline achieves a similar effect by dividing the line into triangles, and then using a texture fetch from a pre-calculated coverage map texture to determine the coverage value of each vertex. The code to perform the texture fetch is inserted at the end of the fragment shader.

5.2.2 Triangle setup:

Triangle setup is responsible for dividing the triangles into bins and setting up attribute coefficient values the will later be used by the GPU to perform attribute interpolation. For each triangle, first the LLVM setup program is executed in order to calculate the attributes’ derivatives (in both X and Y directions), as well as the attributes’ value at the origin. Then, a bounding box test is performed to determine which tiles are touched by the triangle, and then a render command is added to the corresponding bins. In addition to the attribute coefficients and setup planes each command contains a hint to help rasterize the triangle more efficiently. More specifically, for small triangles, the size of the largest bounding sub-tile (4x4, 16x16 or 64x64) is passed, and for large triangles, a flag is passed to indicate whether the 64x64 tile is fully contained by the triangle.

Furthermore, triangle setup is responsible for updating the GPU state. Every time a relevant driver state changes, a command is sent to the GPU to update one of the following GPU states: alpha/depth/stencil coefficients, constants buffer, textures info, and the LLVM IR fragment program.
6 GPU Model

The architecture and implementation of a tile-based rendering GPU model that performs rasterization and fragment processing is presented in this section.

6.1 Architecture

The model was designed as highly flexible and parallel multi-core processor that can be easily extended to support different instruction set architectures (ISA), as well as tweak the design parameters to explore architectural tradeoffs. Most of the processing is done using a shader program, as the only required fixed function units are a command dispatcher, texture unit and a

![Diagram of GPU Model](image)

**Figure 13: Software GPU Model**
small rasterization unit. Since each processor core operates independently on a triangle command bin, scaling the number of processors is trivial. Figure 13 displays the detailed architecture of the GPU. The remainder of this section describes in detail the design of each component.

**6.1.1 Command Dispatcher**

A fairly simple unit that is responsible for receiving commands from the driver and controlling the rest of the units. When a render bin command is received, the dispatcher sends it to the next free core in a round robin order. Furthermore, it is responsible for programming the color/depth tile caches and the texture units.

**6.1.2 Core**

The cores are the most important component of the GPU, as they are responsible for rendering the triangles. Each core operates on a tile at a time; therefore, it is guaranteed that only the corresponding color and depth buffers tiles will be touched. As such, when a core receives a new render bin command, it first pre-fetches the associated depth and color tiles into a local cache. All rendering operations are then performed using the cached copy. When the core is done rendering all the triangles in the bin, it writes the cache back into main memory. As a result, both the color buffer and depth buffer values are read and written only once to main memory, regardless of how many overlapping triangles there are in the bin. To further reduce memory traffic, two optimizations are applied. First, the color buffer read is skipped if the first bin command is a color buffer clear. Second, a depth buffer write is eliminated if the depth buffer is in read only mode.

The fragment program operates on 4x4 tiles at a time. However data is usually stored in memory in linear order. In order to improve memory locality, each cached tile is further subdivided into 4x4 tiles that are stored in 4D tile order. Figure 14 correlates the location of each data block (color or depth value) with an address, expressed in hexadecimal. An ‘x’ represents an arbitrary hexadecimal number. The left block shows the layout of a 4x4 image in 2D order. A 4x4 grid of these 2D blocks gives rise to a 16x16 image laid out in 4D order as illustrated in the right block. The first hexadecimal digit determines the block, and the second hexadecimal digit determines the pixel within that block.
Furthermore, the fragment program operates on SOA data, while the color buffer is stored in memory in AOS format. Hence, the color tile cache also performs an AOS to SOA conversion when reading the color buffer tile, and SOA to AOS conversion when writing the color buffer tile as shown in figure 15.

Figure 14: Tiling Algorithm

The rasterizer operates on a single bin of triangles at a time. Each triangle gets rasterized to determine which pixels it covers, resulting in 4x4 fragment tiles with a coverage mask for each pixel. A hierarchyal rasterization algorithm is implemented as follows. First the tile that is assigned to the core is divided into 16x16 sub-tiles, and each sub-tile is tested against the triangle to determine if it is: completely outside the triangle, completely inside the triangle, or touches the triangle. The above coverage test can be easily performed by comparing the 4 corners of the tile against the triangle’s line equations. If the tile is completely inside the triangle, it gets passed to the fragment program. If the tile touches the triangle it gets divided into 4x4 tiles and the above test gets repeated. The test is repeated once more for 4x4 tiles that touch the triangle in order to calculate the coverage mask for individual fragments.

Some of the above stages can be optimized out using the information provided by the driver. If the smallest bounding box for the triangle is known, the first one or two stages can be skipped. If it is known that the tile if fully inside the triangle, rasterization is skipped entirely.
The shader processor executes the fragment program for each 4x4 tile. There are two fragment program versions, one to execute when the 4x4 tile is fully inside the triangle, and one to execute when a coverage mask is passed to determine which of the fragments in the tile are inside the triangle.

Each core can have multiple threads running at the same time, where each thread executes the shader program on a different 4x4 tile. When the program reaches a texture call, the thread must wait for the result from a texture unit. Instead of stalling, the core then switches to a different execution thread.

6.2 GPU Model Implementation

As described in section 4.2, our design uses LLVM IR as the default ISA; the driver sends the GPU a LLVM IR program to execute for every 4x4 fragment stamp. In order to implement a fully functional GPU, one needs to extend the above architecture with a processor backend that can execute the LLVM IR shader code provided by the driver. First, a baseline model that executes raw LLVM IR using the LLVM runtime was created. Then, to demonstrate the competence and flexibility of the architecture, a second backend that uses MVM IR as an ISA was created.

6.2.1 LLVM Backend

A simple LLVM backend was implemented using LLVM runtime. Each LLVM IR program is parsed, and then executed using LLVM JIT. For more information about LLVM runtime and JIT is available in the LLVM documentation [33].

6.2.2 MVM Backend

MVM IR is a proprietary ISA used by Metabacus Inc. for its suite of behavioral synthesis tools. It comes with a set of tools that allow emulating the IR by compiling it into a C program. As shown in Figure 16, in order to execute the input LLVM IR program; first the LLVM IR program is translated into MVM IR; then the MVM IR program is translated into a C program using an MVM-CGEN tool; and finally, the C program is compiled into shared object file.
One of the reasons for choosing LLVM IR as the shader IR for this thesis was the reusable design of the LLVM framework and the ease of writing new LLVM backends for translating LLVM IR into various target ISAs. LLVM provides two methods for writing a backend:

1. LLVM Code Generator Framework[34], a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target. More specifically it provides tools such as: abstract machine code, target independent compiler algorithms (register allocation, scheduling etc.) and JIT support.

2. Writing a custom code generator by directly translating LLVM IR instruction into target ISA. The LLVM C backend is one such example.

Since MVM IR was modeled after LLVM IR, and is closer to it than to actual hardware ISAs, the latter approach was taken.
7 Architectural Exploration

The main goal of this thesis was to build a framework that can be used for designing and evaluating the performance of a software-based GPU. We demonstrate the completeness of this framework by conducting a series of performance and scaling studies. We present the results of these studies bellow.

7.1 Workload and Simulation Method

All the tests were conducted using 4 well known benchmarks from the Phoronix Test Suit [35]: Quake, Unreal Tournament, Padman and Light. For each benchmark 10 frames where taken at intervals of 25 frames. We captured the frames using Gallium 3D trace driver, and first replayed them using Gallium3D softpipe driver to ensure that the right images were produced. Then we used the driver MVM GPU model to play back the traces.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Resolution</th>
<th>Frame Rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unreal</td>
<td>640x480</td>
<td>32</td>
</tr>
<tr>
<td>Quake</td>
<td>640x480</td>
<td>32</td>
</tr>
<tr>
<td>Light</td>
<td>1024x768</td>
<td>32</td>
</tr>
<tr>
<td>Padman</td>
<td>1024x960</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 1: Benchmarks

Our framework allows us to tweak several variables:

- Tile size: The size of each tile in the binning algorithm. Each benchmark was executes with 8x8, 16x16, 32x32, and 64x64 tile size.

- Number of cores: The number of cores in the GPU. Each benchmark was executed with 1, 2, 4, and 8 cores.

- Number of thread: The number of simultaneous threads supported by each core. Each benchmark was executed with 1, 2, 4, and 8 threads per core.
• Texture Latency: The latency in cycles of a texture access made by the shader. Each benchmark was executed with 300 and 500 cycles texture latency.

• Asynchronous states: The ability for each core to maintain its own state as opposed to sharing one state between all cores. When all cores share the state, each core has to wait for all other cores to finish execution before it can start executing a batch of triangles with a new state. Each benchmark was executed with asynchronous states on (each core has its own state), and off (all cores share one state).

Each benchmark was executed with a permutation of the above options for a total of 16,384 executions per benchmark. For each execution the following performance, utilization and load balance indicators were collected.

• Total number of cycles: The number of cycles it took the GPU to complete rendering all the frames

• Total number of triangle: The number of triangles rendered by the GPU

• Total number of bins: The number of bin rendered by the GPU

• Average batch utilization: The average ratio of cycles spent doing work for each group/batch of thread executed; the rest of the time is spent waiting for texture operations.

• Average core utilization: The average ration of cycles each core spends doing work as opposed to waiting for other cores.

• Standard deviation of core utilization. Measure the load imbalance between the cores.

7.2 Simulation Speed

The purpose of performing an architectural simulation is to quickly explore the design space of the system and evaluate the pros and cons of architectural decisions. Typically, even the simplest system, will have many architectural variables that can be combined together for a large number of combinations that all need to be tested; in our case, there are a total of 16,384 different
combinations. As such, for a simulator framework to be practical in real world settings, and to allow rapid prototyping of a system, it has to be reasonably fast.

Typical GPU simulators, such as Attila\cite{13}, are (a) cycle accurate and (b) use an instruction level simulator for shader code. While Attila didn’t publish simulation speed results, such systems usually take a number of minutes to execute each frame. Our system achieves an order of magnitude faster simulation speed by compiling all shaders into native instruction. Furthermore, since we translate most pipeline stages into shader code, only a small part of the pipeline is simulated in software. Table 2 shows the simulation speed of 10 frames for each of the benchmarks we used. A noted, executing these benchmarks on a cycle accurate simulator would take an order of magnitude longer.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FPS (Frames per second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unreal</td>
<td>2.29</td>
</tr>
<tr>
<td>Quake</td>
<td>3.52</td>
</tr>
<tr>
<td>Light</td>
<td>0.59</td>
</tr>
<tr>
<td>Padman</td>
<td>1.37</td>
</tr>
</tbody>
</table>

Table 2: Simulation Speed

7.3 Number of Cores Effect on Performance and Load Balancing

Just as commercial GPUs, our GPU can have multiple cores, each with its own local memory and texture unit that can operate on different tiles independently. Under ideal conditions, scaling the number of cores linearly will result in a linear performance gain; however, in practice, this cannot be achieved for a large number of cores for two reasons. First, bins have a varying number of triangles, resulting in some tiles getting processed faster than others. Second, in some architecture, all cores share the same global state: constants, shader programs, blending coefficients, etc. Every state change effectively becomes a synchronization point as the cores have to wait for each other to finish processing all previous bins before starting to render the bins associated with the new state. This often leads to load imbalance, where some cores have more work than others. In an extreme case, if there are more cores than bins between consecutive state changes, some cores will be left with no work to do.

Table 3 shows the effect of increasing the number of cores on load balance. We measure the average core utilization - the percentage of cycles each core was busy doing work; as well as the
standard deviation of the utilization – the deviation between the total number of busy cycles for each core. Quake and Padman exhibit almost perfect load balance even with 8 cores, thereby leading us to the conclusion that both have very few state changes. Unreal and Light, on the other hand, display larger load imbalance as we increase the number of cores. Interestingly, while Unreal has more load imbalance than Light, the former has a higher standard deviation. One hypothesis explaining this behaviour is that Unreal has a lot of bins between consecutive state changes, resulting in different core stalling each time. While Light has several consecutive stage changes with very few bins in between, thereby resulting in the last core stalling every time.

<table>
<thead>
<tr>
<th></th>
<th>Number of Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quake</strong></td>
<td>Total Cycles</td>
<td>1708768706</td>
<td>854409229</td>
<td>427229197</td>
<td>213645418</td>
</tr>
<tr>
<td></td>
<td>Avg Core Utilization</td>
<td>100%</td>
<td>99.9%</td>
<td>99.9%</td>
<td>99.9%</td>
</tr>
<tr>
<td></td>
<td>Core Utilization StDev</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Unreal</strong></td>
<td>Total Cycles</td>
<td>1430157720</td>
<td>748869452</td>
<td>430461866</td>
<td>2.88E+08</td>
</tr>
<tr>
<td></td>
<td>Avg Core Utilization</td>
<td>100%</td>
<td>95%</td>
<td>83%</td>
<td>62%</td>
</tr>
<tr>
<td></td>
<td>Core Utilization StDev</td>
<td>0</td>
<td>0.004</td>
<td>0.018</td>
<td>0.038</td>
</tr>
<tr>
<td><strong>Padman</strong></td>
<td>Total Cycles</td>
<td>2562917448</td>
<td>1281832185</td>
<td>641015588</td>
<td>320705833</td>
</tr>
<tr>
<td></td>
<td>Avg Core Utilization</td>
<td>100%</td>
<td>99.9%</td>
<td>99.9%</td>
<td>99.9%</td>
</tr>
<tr>
<td></td>
<td>Core Utilization StDev</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Light</strong></td>
<td>Total Cycles</td>
<td>41250404038</td>
<td>22796566898</td>
<td>11842830857</td>
<td>6098991984</td>
</tr>
<tr>
<td></td>
<td>Avg Core Utilization</td>
<td>100%</td>
<td>99%</td>
<td>95%</td>
<td>92%</td>
</tr>
<tr>
<td></td>
<td>Core Utilization StDev</td>
<td>0</td>
<td>0.011</td>
<td>0.028</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table 3: Statistics from varying the number of cores while keeping all other caribale constant at 64x64 tiles, 1 threads/core, 500 cycles texture latency and async states on

Figure 17 shows a graph of the performance achieved as we increase the number of cores. Predictably, the performance of Quake and Padman scales linearly with the number of cores, while Unreal and Light show a slower performance gain. Increasing the number of cores clearly gives a significant performance boost; however, it also comes at a high cost in terms of chip area and power. Thereby one should consider both efficiency and performance when making design decisions. For example, Unreal gets a 496% speedup with 8 cores; however the GPU is active only 62% of the time, resulting in waste of chip resources.
Figure 17: The effect of increasing the number of cores on performance, as measured by the speedup compared to a one core configuration. All other variables are kept constant: 64x64 tiles, 1 thread/core, 500 cycle texture latency and async state on.

### 7.4 Number of Threads per Core Effect on Performance and Core Utilization

Each core can have multiple threads active at the same time; when one thread stalls on a texture access, the next thread takes over. The ability of the GPU to hide texture latency depends on two factors: the number of instructions between consecutive texture calls, and the number of concurrent threads. Increasing either one of those factors increases the number of potential instructions the core can execute before being forced to wait for the texture call to return. Table 4 shows the effect of increasing the number of threads on performance and core utilization.
<table>
<thead>
<tr>
<th></th>
<th>Threads</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quake</td>
<td>Total Cycles</td>
<td>427203164</td>
<td>298238559</td>
<td>247131635</td>
<td>243745410</td>
</tr>
<tr>
<td></td>
<td>Average Batch Utilization</td>
<td>0.7</td>
<td>0.81</td>
<td>0.94</td>
<td>0.97</td>
</tr>
<tr>
<td>Unreal</td>
<td>Total Cycles</td>
<td>357554693</td>
<td>268161255</td>
<td>219242243</td>
<td>2.16E+08</td>
</tr>
<tr>
<td></td>
<td>Average Batch Utilization</td>
<td>0.8</td>
<td>0.84</td>
<td>0.95</td>
<td>0.97</td>
</tr>
<tr>
<td>Padman</td>
<td>Total Cycles</td>
<td>518,486,245</td>
<td>382,975,795</td>
<td>325286438</td>
<td>316516874</td>
</tr>
<tr>
<td></td>
<td>Average Batch Utilization</td>
<td>0.64</td>
<td>0.89</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Light</td>
<td>Total Cycles</td>
<td>1126394111</td>
<td>9233814703</td>
<td>8981530663</td>
<td>8921687112</td>
</tr>
<tr>
<td></td>
<td>Average Batch Utilization</td>
<td>0.83</td>
<td>0.93</td>
<td>0.98</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table 4: Statistics from varying the number of threads while keeping all other variables constant at 64x64 tiles, 4 cores, 500 cycles texture latency and async states on.

With only one active thread all benchmarks exhibit performance loss due to texture latency, spending 17%-36% of the cycles stalling. The difference between the benchmarks is due to the difference in the average number of shader instructions between consecutive texture calls. Padman has more texture calls, and less arithmetic shader operations compared to the other benchmarks, while Light has the lowest ratio of texture call to arithmetic calls.

Figure 18 shows a graph of the speedup of each benchmarks as we increase the number of threads. It can be seen that the speedup for Quake, Unreal and Padman starts saturating at 4 threads, while the speedup of Light stats saturating at two threads. Hence, it can be concluded that for the benchmarks studied in this thesis, and for a GPU with 500 cycles texture latency, there is little value in having more than 4 simultaneous threads per core.
In the previous section we studied the effects of changing the number of simultaneous threads per core when the texture latency is 500 cycles. As mentioned earlier, the number of threads required to hide texture latency linearly depends on the actually latency. Hence, in this section we further investigate the topic by varying both texture latency and the number of threads. Table 5 shows the effect of increasing the number of threads on performance and core utilization for 300 cycle texture latency as well as 500 cycles.

Figure 18: The effect of increasing the number threads/core, as measured by the speedup compared to a one thread/core configuration. All other variables are kept constant: 64x64 tiles, 4 cores, 500 cycle texture latency and async state on

7.5 Texture Latency Effect on Performance

In the previous section we studied the effects of changing the number of simultaneous threads per core when the texture latency is 500 cycles. As mentioned earlier, the number of threads required to hide texture latency linearly depends on the actually latency. Hence, in this section we further investigate the topic by varying both texture latency and the number of threads. Table 5 shows the effect of increasing the number of threads on performance and core utilization for 300 cycle texture latency as well as 500 cycles.
Table 5: Statistics from varying the texture latency and number of threads while keeping all other variables constant at 64x64 tiles, 1 core, and async states on.

Figure 19 shows a graph of the performance gain when the texture latency is 300 cycles instead of 500 cycles. It can be seen that when there is only one thread, we get a 9%-24% gain in performance due to the fact that there are no extra threads to hide the texture access latency.

### 7.6 Tile Size Effect on Performance and Core Utilization

The driver splits the triangles into bins (associated with tiles). When a triangle covers more than one tile, it gets added to the bins for all the tiles it covers; which in turn results in it being rasterized multiple times by the GPU. The smaller the tile, the higher are the chances of a triangle covering multiple tiles. When reducing the tiles size by a factor of two (i.e. 64 to 32) the original tile gets subdivided into four new tiles; in the worst case scenario, when the scene consists of very large triangles, each triangle touches all sub-tiles, and ends up being rasterized four times. However, since the rasterizer is smart enough to pass the shader only the fragments inside the tile being processed, the overhead is incurred only by the rasterizer – the shader processes the same number of fragment regardless of the tile size.
Figure 19: The speedup achieved by reducing the texture latency from 500 cycles to 300 cycles on performance for various number of threads/core. All other variables are kept constant: 64x64 tiles, 1 core and async state on.

Table 6 shows the effect of varying the tile size on the number of processed tiles and triangles. One would expect that when reducing the tile size by a factor of four (i.e. from 64x64 to 32x32), the total number of rendered bins will increase by a factor of 4; however, our results show an increase by a factor of only 2.36 – 3.98. This can be explained by the fact that some of the smaller sub-tiles are not touched by any triangles and hence never get processed. Furthermore, we can see that the number of rendered triangles increase by a factor of 1.45 – 2.08 when halving the tile size. However, since our study focuses on shader performance, and ignores rasterizer performance, the extra triangles have no effect on performance.

Reducing the tile size also reduces the average number of triangles and fragment quads per bin. If a bin has an insufficient number of quads, the core cannot keep all of its threads busy, thereby reducing its ability to hide texture latency. Table 6 also shows the core utilization of a GPU with 1 core and 8 threads for different tile sizes; as we increase the tile size the core utilization increases slightly. Figure 20 shows a graph of the speedup of the various benchmarks as we increase the tile size. We can see a correlation between the increase in batch utilization and the increase in performance.
<table>
<thead>
<tr>
<th></th>
<th>Tile size</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quake</td>
<td>Total Cycles</td>
<td>1289332759</td>
<td>1274303011</td>
<td>1267866966</td>
<td>1264937921</td>
</tr>
<tr>
<td></td>
<td>Total Triangles</td>
<td>191752</td>
<td>96967</td>
<td>53987</td>
<td>35502</td>
</tr>
<tr>
<td></td>
<td>Total Bins</td>
<td>1222679</td>
<td>325074</td>
<td>93853</td>
<td>34096</td>
</tr>
<tr>
<td></td>
<td>Avg Batch Utilization</td>
<td>95%</td>
<td>96.1%</td>
<td>97.6%</td>
<td>98.4%</td>
</tr>
<tr>
<td>Unreal</td>
<td>Total Cycles</td>
<td>1411450234</td>
<td>1396526568</td>
<td>1376526568</td>
<td>1361526568</td>
</tr>
<tr>
<td></td>
<td>Total Triangles</td>
<td>157078</td>
<td>75259</td>
<td>37514</td>
<td>21295</td>
</tr>
<tr>
<td></td>
<td>Total Bins</td>
<td>291976</td>
<td>83626</td>
<td>27259</td>
<td>11524</td>
</tr>
<tr>
<td></td>
<td>Avg Batch Utilization</td>
<td>96.8%</td>
<td>97.4%</td>
<td>98.1%</td>
<td>99.0%</td>
</tr>
<tr>
<td>Padman</td>
<td>Total Cycles</td>
<td>3082649628</td>
<td>3082366450</td>
<td>3081627248</td>
<td>3081671705</td>
</tr>
<tr>
<td></td>
<td>Total Triangles</td>
<td>19535</td>
<td>9802</td>
<td>4821</td>
<td>2323</td>
</tr>
<tr>
<td></td>
<td>Total Bins</td>
<td>582772</td>
<td>146176</td>
<td>36793</td>
<td>9287</td>
</tr>
<tr>
<td></td>
<td>Avg Batch Utilization</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>Light</td>
<td>Total Cycles</td>
<td>61582016497</td>
<td>60752678890</td>
<td>60698132631</td>
<td>60643120606</td>
</tr>
<tr>
<td></td>
<td>Total Triangles</td>
<td>772182</td>
<td>408696</td>
<td>241823</td>
<td>166314</td>
</tr>
<tr>
<td></td>
<td>Total Bins</td>
<td>1084047</td>
<td>312541</td>
<td>100807</td>
<td>38287</td>
</tr>
<tr>
<td></td>
<td>Avg Batch Utilization</td>
<td>96.9%</td>
<td>97.8%</td>
<td>98.2%</td>
<td>98.9%</td>
</tr>
</tbody>
</table>

Table 6: Statistics from varying the tile size while keeping all other variables constant at 1 core, 8 threads/core, 500 cycles texture latency and async states off

Figure 20: The effect of increasing the tile size on performance, as measured by the speedup compared to an 8x8 tile configuration. All other variables are kept constant: 64x64 tiles, 1 thread/core, 500 cycle texture latency and async state on.
8 Conclusion

This thesis describes the design of a software based GPU framework. This framework is built with the idea that eliminating most hard-coded GPU blocks, and running the entire 3D pipeline using shader programs is aligned with the current trend towards heterogeneous computing. Such design has the potential to increase 3D graphic system efficiency, reduce power consumption, decrease bring-up time, and open the door for new 3D graphic effects and algorithms.

Using LLVM as an intermediate shader representation is a one key to our design as it provides flexibility, and allows us to easily retarget new GPU instruction set architectures (ISA). We describe some of the novel aspects of our use of LLVM, including implementation of hardware units in LLVM IR, shader code translation and execution. Another key aspect of our design is the tile-based, sort middle, architecture. We describe the benefits of using tiles, including: reduced memory traffic and power consumption, better caching performance, and ease of scaling the number of cores.

One important contribution of this thesis is the implementation of the design. We build a driver, as well as two separate GPU models supporting LLVM IR and MVM IR as shader ISA. We proceed by showing how the framework can be used to perform an architectural exploration of the design space in order to evaluate design tradeoffs.

The software GPU architecture framework presented in this paper is a complete system with a driver and GPU model that can execute standard 3D games and benchmarks. The system features a modular design that allows easily designing, building and testing new features and GPU ISA. We look forward to seeing how future efforts can use our system to design new, software based, GPU hardware.
References


   http://www.highperformancegraphics.org/previous/www_2010/media/Hot3D/HPG2010_Hot3D_ARM.pdf


[18] Chris Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization,".


  http://www.nvidia.ca/content/tesla/pdf/NV_DS_TeslaK_Family_May_2012_LR.pdf

[23] RightWare, "Benchmarking Software," [Online].


[41] Samsung, "Enjoy the Ultimate WQXGA Solution with Exynos 5 Dual,". [Online].
http://www.samsung.com/global/business/semiconductor/minisite/Exynos/data/Enjoy_the_Ultimate_WQXGA_Solution_with_Exynos_5_Dual_WP.pdf

http://llvm.org/docs/LangRef.html

