Teaching Graphics Processing and Architecture using a Hardware Prototyping Approach

Michael Steffen, Phillip Jones, and Joseph Zambreno
Electrical and Computer Engineering
Iowa State University, Ames, IA, USA
{*steffma, phjones, zambreno*}*@iastate.edu*

***Abstract*—Since its introduction over two decades ago, graphics hardware has continued to evolve to improve rendering performance and increase programmability. While most undergraduate courses in computer graphics focus on rendering algorithms and programming APIs, we have recently created an undergraduate senior elective course that focuses on graphics processing and architecture, with a strong emphasis on laboratory work targeting hardware prototyping of the 3D rendering pipeline. In this paper, we present the overall course layout and FPGA-based laboratory infrastructure, that by the end of the semester enables students to implement an OpenGL-compliant graphics processor. To our knowledge, this class is the first that takes a hardware prototyping approach to teaching computer graphics and architecture.**

I. Introduction

The Graphics Processing Unit (GPU) plays a fundamental role in all personal computing, from the largest workstations to portable multimedia-enabled embedded systems. The highest-capacity GPUs are larger (more transistors), and have higher performance (in terms of GFLOPs) than conventional multi-core CPUs. The recent emergence of General-Purpose computation on GPU (GPGPU) computing paradigm has increased the relevance of GPU architecture to students in electrical and computer engineering and their potential employers.

Courses in computer graphics are commonly-found in EE/CS/CE curricula [1], [2], [3], including more recently introduced courses on programmable parallel GPU architectures [4]. However, these courses tend to focus on graphics from the viewpoint of algorithms, modeling, or software development. Architectural issues are covered for the sake of completeness and for understanding performance implications, but laboratory exercises tend to be software-only (e.g. programming in OpenGL, DirectX, CUDA, OpenCL).

At Iowa State University, we have recently developed a senior elective class that introduces computer graphics from the perspectives of the hardware architect, system designer, and software programmer. Students (with no prior background in computer graphics) are presented with a historical view of the challenges and innovations that have evolved graphics processing from simple frame-buffer manipulation, to 2D rendering, to 3D rendering, to current unified shader architectures for rendering and GPU computing. By the end of the semester, students are able to design hardware pipelines for several varieties of GPU architecture, can modify an existing graphics API to interface with an FPGA-based development platform, and have gained familiarity with writing highly multi-threaded GPU software for graphics rendering and general-purpose computation.

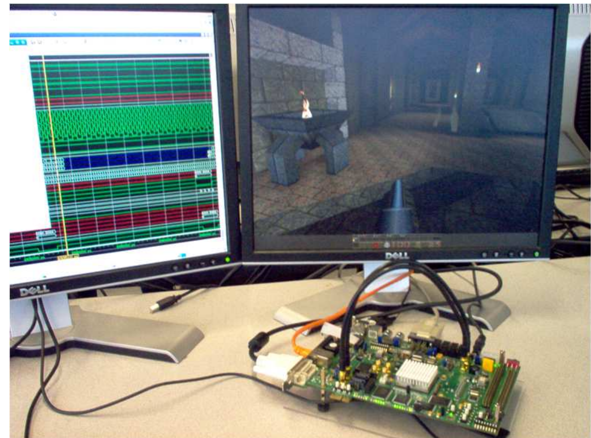

Figure 1. An FPGA board running a 3D fixed-function graphics pipeline. The software engine for the popular video game *Quake* [5] is running on the desktop computer, and OpenGL commands are sent to the FPGA board for rendering

In addition to in-class lectures, weekly laboratories are assigned for students to gain practical experience designing and implementing components of the GPU pipeline. The semester-long goal of the laboratory assignments is for students to prototype a conventional GPU architecture, code-named the Simple Graphics Processor (SGP), capable of running most OpenGL graphical applications. As a special incentive to students, the popular OpenGL-compatible video game *Quake* [5], developed by ID Software, is provided as a benchmark application that they can test from day one. Weekly progress can be evaluated with this application, and only by the end of the semester will their SGP designs be capable of executing all of the OpenGL library calls required to render the game with expected fidelity (see Fig. 1).

II. Laboratory Resources

In the lab, students have access to a Linux-based workstation with all the software tools installed, an FPGA prototyping board, and an FPGA reference design (including software code) for interfacing to the FPGA from the workstation computer.

A. Workstation Setup

An NVIDIA GeForce GTX 480 [6] is installed in every workstation computer for accelerating OpenGL applications and to support the dual-monitor setup. Dual monitors provide several productivity advantages, for hardware design (viewing code and waveforms at once), graphics application debugging (viewing source code and executed graphics application) and when using the FPGA board (viewing the desktop computer screen and the FPGA video output). An NVIDIA GPU also supports CUDA applications that students learn about in lectures and in programming assignments.

B. Laboratory Hardware

Twelve of the workstations in the lab are attached to a Xilinx Virtex-5 FPGA XUPV5-LX110T board [7], which contains a Xilinx Virtex-5 LX110T FPGA, and supports a wide range of useful peripherals for implementing graphics processors, including:

- DVI / VGA video output
- RS-232 serial port
- 10/100/1000 Ethernet PHY interface
- JTAG programming interface
- 256 MB of DDR2 SODIMM memory
- PCI Express x1 connector

The Xilinx Virtex-5 LX110T FPGA contains 110,592 logic cells and a total on-chip memory size of 5,328 Kbits. For our purposes, one limiting factor is the 64 Digital Signal Processing (DSP) slices, which are used to accelerate multiplication and division operations. Since matrix-matrix and matrix-vector multiplication is a common operation in the graphics pipeline, optimizing performance given tight resource constraints becomes an important design challenge for students throughout the semester.

C. Driver and Interfacing

For OpenGL applications to use the SGP hardware, our laboratory infrastructure requires a method for sending commands to the FPGA board. To accomplish this we modified the GLTrace [8] OpenGL `libGL.so` library. When a graphics application calls an OpenGL function, our `libGL.so` library captures that function and sends instructions to the SGP using one of the supported communication methods. At the same time, we want students to see the correct implementation of the OpenGL functions on the workstation monitor. To accomplish this, our `libGL.so` library also loads the native `libGL.so` library that comes installed with the graphics card driver. Our SGP OpenGL driver then forwards the OpenGL calls to the workstation graphics card. Figure 2 shows a diagram of the library structure. Any OpenGL application that can run on the local workstation can then be transparently modified to use the FPGA-based SGP for rendering.

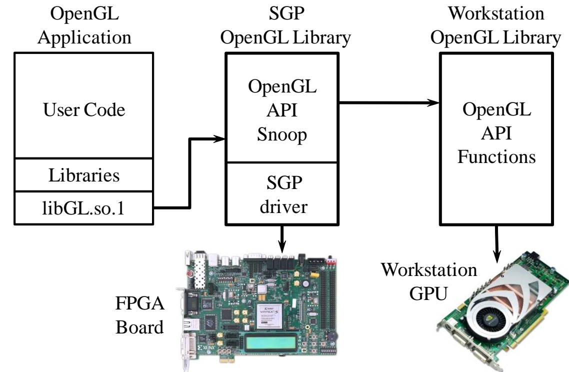

Figure 2. The SGP OpenGL driver sends instructions to the FPGA board and also forwards the OpenGL commands to the desktop OpenGL driver. This setup allows the workstation computer to render the same application as the SGP for comparison.

III. Simple Graphics Processor Architecture

For students to be able to directly begin implementing the core graphics pipeline hardware, basic components of a graphics processor are provided. Provided components include the communication interfaces with the workstation computer, instruction forwarding to sub-processor components, DDR2 memory arbitration, DDR2 controller, and a display controller. Figure 3 shows the architecture of the SGP framework provided to students.

The display controller is configured to use a resolution of 1280x1024. The Xilinx XUPV5 board interfaces with a Chrontel CH7301C chip that can drive both DVI and VGA video. Due to the size of the frame buffer for this resolution, off-chip DDR2 memory is used. The board limits designs to a single DDR2 memory interface and consequently all student-implemented graphics pipeline components that require DDR2 memory must first go through an arbiter. The memory arbiter is designed to be scalable, allowing students the option of adding DDR2 memory interfaces to as many processor components as they see fit.

The SGP currently supports two different communication methods, UART and Gigabit Ethernet[1]. Instructions sent to the SGP are processed by the host block (Host) and are then placed on a communication bus (hostBus) to the targeted sub-component. One sub-component on the communication bus is a Memory Operation (memOps) controller that allows for memory copies and memory sets within the DDR2

[1] PCI Express offers more potential bandwidth than GigE; this is an avenue for planned future work

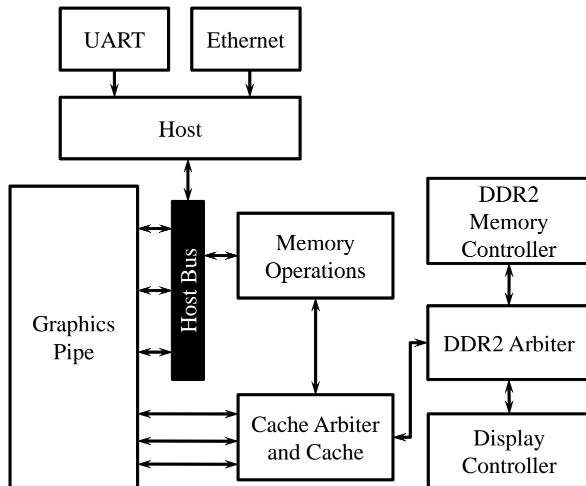

Figure 3. The SGP framework provides the communication method with the workstation computer and sets up the the DDR2 memory interfacing and display controls. Students then implement the graphics pipe component and interface to the framework through specified interfaces.

memory space. This sub-component is also used for sending data from the workstation computer directly to the SGP DDR2 memory.

The Graphics Pipeline (graphicsPipe) component is where students implement the 3D rasterization pipeline. Each pipeline stage can be independently connected to the communication bus for receiving configuration instructions. A communication bus interface component is provided that interfaces with the communication bus signals and stores instructions into a FIFO. The use of the bus interface gives students a standard protocol for receiving pipeline instructions from the host. Direct connections between the individual pipeline stages is left to the students to implement. The DDR2 interface and associated logic operates at 200 MHz, while the graphicsPipe and other student-developed logic is initially constrained to 100 MHz.

IV. Laboratory Assignments

The purpose of the laboratory assignments is to have the students expand the SGP framework described in the previous section to a proper 3D rendering pipeline. The assignments (which we refer to as Machine Problems (MPs) to convey that they have significant hardware as well as software components) are organized such that through the progression of the semester, students can visualize how well their existing SGP processor can support OpenGL applications. We carefully select which OpenGL calls need be supported, so that by the end of the semester the SGP implementations can faithfully render the well-known OpenGL-compatible video game *Quake* [5].

Figure 4 shows the conventional 3D graphics rendering pipeline, and what MPs correspond to each of the hardware pipeline stages.

A. MP-0: Platform Introduction

Students are given two weeks to become familiar with the FPGA hardware design process. While a course in computer architecture and some hardware design experience is a prerequisite, most of the students at this stage will not have had direct FPGA design experience. The assigned tasks include writing VHDL or Verilog code, simulation of the design using Modelsim, and implementation using the Xilinx ISE design suite. MP-0 requires students to implement a processor that performs matrix-vector multiply-accumulation. The students are guided through the process of laying out an architecture on paper, implementing the code in VHDL / Verilog, simulation, and FPGA-based implementation. These tasks prepare students for future MPs and creates a component that can be reused in MP-2.

B. MP-1: Pixel Processing

The SGP framework is introduced in MP-1. Students are guided through the important parts of the HDL code and software / libraries that they will be required to interface with. In addition, the lab requires students to describe implementation methods and design decisions to understand different implementation options and their tradeoffs. Once students have an understanding of the SGP framework, the lab requires them to implement the hardware to draw pixels to the framebuffer and implement the SGP OpenGL driver code to clear the framebuffer. By the end of the lab, students should have a strong understanding of how data is passed from the OpenGL application to the SGP framework, and from the framework to the components that comprise the graphics pipeline.

C. MP-2: Vertex Transformations

Transforming 3D points to screen coordinates requires multiplying 4x1 vectors of 3D points by three different 4x4 matrices and performing perspective division. All math operations use fixed-point notation due to constraints on the amount of FPGA computation resources. A major learning objective for this MP is understanding fixed-point arithmetic and maintaining fractional precision. MP-2 also challenges students to design and implement the required matrix-matrix and matrix-vector multiplication, while utilizing only four 64-bit multipliers and no more than half of the FPGA's DSP slices.

D. MP-3: Generating Fragments

The process of creating fragments (pixels) involves assembling points into triangles and then rasterizing the triangle to fill all the pixels inside the triangle. The first part of MP-3 requires students to assemble triangles, since the points that are streamed in can take on many different formats. Students are then required to buffer certain points depending on the drawing mode and assemble 3-point triangles for rasterization. During rasterization, pixel fragments are

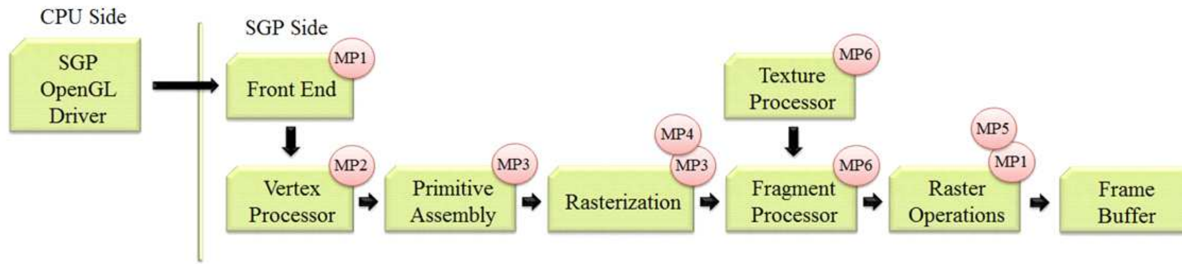

Figure 4. Conventional 3D rasterization pipeline and the Machine Problems (MPs) that correspond to each pipeline stage

generated that are inside the triangle. Students are provided with an algorithm explanation, developed specifically for FPGA implementation, that uses a z-scan fill method. The performance optimization goal is to generate one fragment every clock cycle.

E. MP-4: Color Interpolation

Once fragments have been generated, their color value and depth value must be interpolated using the triangle's three defining points. A total of five interpolations are required, one for each of the four color channels (red, green, blue and alpha), and one for the depth. Students are encouraged to design one hardware component that can be reused for all five interpolation tasks.

F. MP-5: Depth Testing

The framebuffer contains several distinct memory regions used during the rendering processes. Two commonly-used regions for rendering are a color buffer and a depth buffer. The depth buffer keeps track of the depth value (z dimension) for each pixel. When drawing colors to the color buffer, the depth value for that pixel location can be checked to see if the current pixel should be overwritten if it is in front of the previous pixel at its location. In MP-5, students are responsible for managing the memory space and implementing the hardware to handle reading and writing from both the color and depth buffer, while minimizing DDR2 memory traffic.

G. MP-6: Shading and Textures

Applying images to the surface of rasterized triangles adds additional realism to the rendering process. The goal of MP-6 is to implement a texture processing unit for performing the operations required to identify the color for a pixel when given an image coordinate location. Since the texture coordinates do not directly match the dimensions of the texture image, different filtering methods may be implemented to determine the final pixel color. Due to semester length restrictions, we only require students to implement the nearest texture pixel value as the final fragment color. Once students have finished implementing MP-6, *Quake* is close to being fully supported (it still requires alpha blending) by the SGP running on the Xilinx XUPV5 FPGA.

H. MP-7: Pipeline Optimization

The final laboratory assignment allows students to profile their SGP and determine bottlenecks. Students can then explore different options for reducing the bottlenecks and improving the performance of their rendering pipeline. At the end of the semester, a `Quake` tournament is held, with students who have better optimized their FPGA implementation being at a competitive advantage (in terms of rendering latency and throughput).

V. Conclusion

The course presented in this paper offers students a unique perspective to graphics processing and architecture. The individual laboratory assignments and resources allow students to complete a hardware prototype of an OpenGL-compliant GPU running on an FPGA board. Through this class, undergraduate students in our department gain advanced experience with applying computer architecture and hardware design concepts to the graphics processing domain.

References

[1] Jason Lawrence, University of Virginia, *CS 4810*, Available: http://www.cs.virginia.edu/ gfx/Courses/2010/IntroGraphics

[2] David Luebke, University of Virginia, *CS 446*, Available: http://www.cs.virginia.edu/ gfx/courses/2004/RealTime

[3] Kurt Akeley and Pat Hanrahan, Stanford University,*CS448A*, Available: http://graphics.stanford.edu/courses/cs448a-01-fall

[4] Sanjay J Patel, University of Illinois, *ECE 498 AL*, Available: http://courses.engr.illinois.edu/ece498/al

[5] ID Software, *Quake*. http://idsoftware.com/games/quake/quake

[6] NVIDIA Corporation, *GeForce GTX 480*, Available: http://www.nvidia.com/object/product_geforce_gtx_480_us.html

[7] Xilinx Inc, *XUPV5-LX110T Development Board*, Available: http://xilinx.com/univ/xupv5-lx110t.html

[8] Phil Frisbie Jr, *GLTrace*, Available: http://hawksoft.com/gltrace