FPGA-BASED 3D GRAPHICS PROCESSOR WITH PCI-BUS INTERFACE, AN IMPLEMENTATION CASE STUDY

Hans Holten-Lund
Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark
Phone: +45 4525 3351 , Fax: +45 4593 0074
hahl@imm.dtu.dk

ABSTRACT

Case study of an FPGA implementation of a 3D graphics processor. Practical design issues dealing with a PCI-bus based reconfigurable FPGA prototyping board are discussed. PCI drivers and bandwidth issues are discussed. An analog VGA video output is presented as a solution to some of the bandwith issues.

1. INTRODUCTION

This paper is a case study covering practical design issues in the design of an FPGA based 3D graphics processor, based on the work done on the Hybris graphics architecture in [4] which describes the design of the algorithms and architecture in greater detail. This paper focuses on practical design issues dealing with the PCI bus and the FPGA prototyping board.

Working hardware/software codesign implementations of Hybris for standard cell design based ASIC (simulated) [3] and FPGA technologies [4, 6] have been demonstrated. The hardware part of the design was carried out using manual translation of the software C source code to a synthesizable VHDL specification. The FPGA implementation was the first physically working hardware implementation of the Hybris renderer back-end. Currently the FPGA implementation shows great potential for future development, for example many of the parallel back-end architectural concepts demonstrated [4] on a multiprocessor PC may be implemented. Additionally a fully functional VGA video output interface was implemented in the FPGA, eliminating the need to transfer the final rendered frames back to the host PC.

2. FPGA IMPLEMENTATION

Based on the work done for the ASIC implementation, an FPGA implementation [4, 6] was made, targeted for a Xilinx Virtex XCV1000 FPGA [8], a modern FPGA which provides a design space equivalent to one million system gates. The tile rendering engine shown in figure 1 is implemented in the FPGA. However this is only one part of the Hybris architecture, in this implementation the remaining front-end processing is done in software on the host PC. In order to fit the tile rendering engine of Hybris onto the FPGA, several changes had to be made to the original design. One of the changes was to reduce the FIFOs between the rendering pipeline stages in the ASIC design, as large FIFOs are excessively expensive to implement in an FPGA.

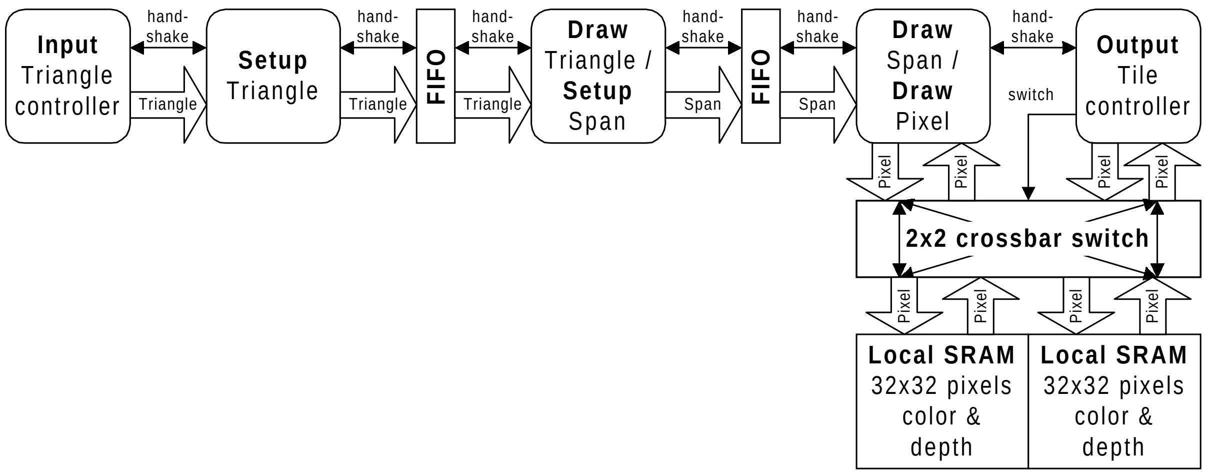

Figure 1. Architectural overview of the tile rendering engine.

The area is much better spent for applying parallelism in the tile rendering engine together with some shorter FIFO buffering. Currently the FIFOs are reduced to a depth of one.

One of the most important changes made was that the on-chip SRAM architecture for on-chip tile buffering was redesigned. The ASIC implementation used a generic Synopsys DesignWare SRAM block, which features an asynchronous *reset* signal to clear the contents of the entire SRAM. While convenient, this reset behaviour adds complexity to an SRAM making it larger and slower. Further, to implement SRAM blocks in the FPGA we are forced to use the configurable blocks of dual ported SRAM present in the FPGA (known as BlockRAM), which do *not* feature asynchronous reset to clear the contents. VHDL components for the BlockRAMs can be generated by the Xilinx Core Generator's dual port SRAM generator. A properly pipelined tile buffer architecture using this more area efficient type of on-chip RAM is shown in figure 1. Here double-buffering and cross-bar switching is used to allow multiple accesses to the dual ported 32x32 pixel tile buffers, providing an efficient tile buffer memory architecture which also allows enhanced rendering algorithms to read pixels, which is needed for alpha blending operations and other advanced rendering methods, such as those discussed in [2].

The tile depth *z*-buffer is now double buffered allowing it to be cleared and stored in a global depth-buffer for future use if required. In comparison, the ASIC implementation relied on asynchronous clear with a single buffered depth buffer.

Additionally it is not possible to transfer the complete triangle heap to the FPGA, as it does not have enough on-board memory to accommodate the potentially huge triangle heap. E.g. 64 Mbytes are needed to store a triangle heap for one million rendered triangles. Because of this the PC host software for the FPGA implementation must send triangle buffers for one 32x32 pixel tile at a time to the FPGA board.

The FPGA implementation is targeted for a codesign prototyping platform based on an ordinary Pentium III PC with an FPGA prototyping board on the PCI-bus. This prototyping board is the Celoxica / Embedded Solutions Ltd. RC-1000PP PCI board featuring a PLX PCI 9080 PCI interface and a Xilinx Virtex XCV1000-6 FPGA as well as 8 Mbytes of asynchronous SRAM in four independently addressable 32 bits wide banks. Unfortunately such a prototyping system is limited by the maximum speed of the PCI bus. The available drivers for accessing the RC-1000PP through the PCI-bus limits the bandwidth further. The Xilinx Virtex FPGA architecture is described in [8], and the RC-1000PP manual [7] describes how the FPGA's pins are connected to the other resources on the prototyping board.

The design flow for the FPGA implementation is to start from a suitable reference C implementation and manually transform it into an RTL description of the design in VHDL suitable for FPGA implementation. In this particular study some of the VHDL code developed for the ASIC implementation could be reused. Synopsys FPGA Compiler II is then used to synthesize the VHDL description, targeted for the Virtex FPGA architecture. Once synthesized an EDIF net-list is exported from Synopsys. This EDIF net-list is then imported into the Xilinx Alliance Design Manager FPGA layout tool to map the design to a particular FPGA using placement and routing to finally create an FPGA configuration bit-file. Finally, this bit-file is then used to configure the FPGA to form the designed digital system.

3. PCI BANDWIDTH

The performance of the first FPGA implementation was severely limited by the available PCI bandwidth. The bandwidth problems were mainly caused by poor utilization of the limited communication models available in the driver for the RC-1000PP FPGA prototyping board.

The protocol used in [6] for transferring data over the PCI-bus performed a host-initiated PCI bus-master DMA transfer for each triangle node buffer in the bucket sorted triangle heap. While this type of transfer, once running, will transfer data at maximum speed over the PCI-bus, there is a relatively large overhead for initializing the data transfer. Since the blocks to be transferred are 4 kbytes or less in size, the accumulated overheads of many small transfers becomes very high. In [6] a plot of block size and measured transfer rate is presented, showing a relatively poor transfer rate of about 10 Mbytes/s for continuous transfer of 4 kbyte data blocks. This should be compared to a data transfer rate of over 100 Mbytes/s achievable for a single large block transfer. It should be noted that the actual transfer rate depends on the PCI implementation of the PC's motherboard chip-set: Observed maximum transfer rates for transfers from the PC to the PCI board varied between 50 Mbytes/s with the Intel 815E chip-set, 80 Mbytes/sec with the VIA 694 chip-set and 110 Mbytes/sec with the Intel 440BX chip-set.

According to the PLX PCI 9080 manual [5] the PCI interface on the FPGA board is also capable of performing scatter/gather or chaining mode PCI bus-master DMA transfers. Chaining mode DMA would be the ideal method for transferring data to the FPGA prototyping board, because it is able to match the data distribution in the triangle heap. The bucket sorted triangle heap contains a set of triangle node buffers for each bucket corresponding to a tile. These buffers are not necessarily placed in contiguous memory locations. However, by using chaining mode DMA it would be possible to instruct the PLX PCI 9080's DMA engine to transfer the relevant blocks. In practice this can be done by creating a linked list in either host (PC) or local (FPGA board) memory which is a list of pointers to variable sized memory blocks to be fetched. Once started, the PLX chip automatically executes the transfers by following the links in the list, avoiding the expensive overhead of starting the individual transfers from the host. Unfortunately the supplied RC-1000PP device driver API [1] for Windows only exposes a simplified "2D" interface to this scatter/gather functionality, requiring that the memory blocks are equal in size and spaced equally in memory. Since no source code was supplied for the RC-1000PP API, adding the proper functionality would require starting from scratch with the implementation of a Windows device driver.

Another alternative data transfer method is to use memory mapped I/O. As the FPGA board's driver maps the on-board memory to a virtual address space in the host PC, the FPGA board memory can be accessed as normal memory, although at a lower speed. An implementation of the triangle data transfer based on memory mapped I/O improves the bandwidth to about 30 Mbytes/s. Although better than the original 10 Mbytes/s, it is still far from the maximum PCI bandwidth. The slow speed is caused by address/data multiplexing which halves the available bandwidth, as well as the use of individual PCI bus transfers for each data word.

The best transfer method to the FPGA board was found to be DMA transfers of large data blocks. However this requires re-organization of the 4 kbyte triangle heap memory buffers into larger 2 Mbyte memory blocks which as mentioned earlier can be transferred at the maximum transfer rate of the PCI bus, i.e. up to about 100 Mbytes/s. The re-organization causes a small overhead from memory-to-memory copying, but nevertheless this method is currently the best

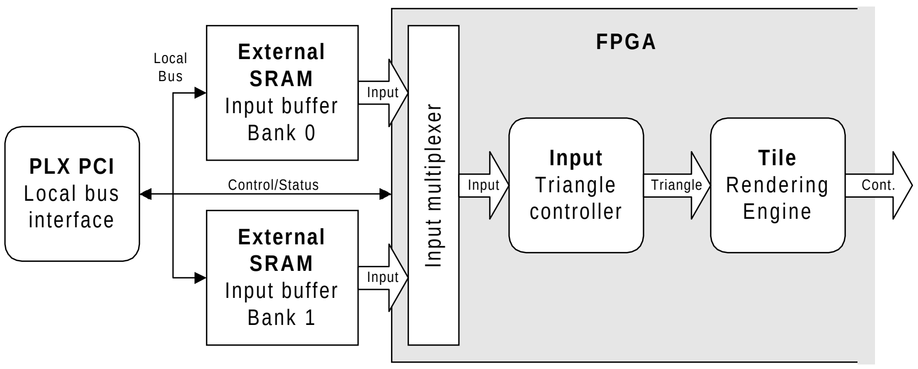
FPGA
Local Bus
External SRAM Input buffer Bank 0
External SRAM Input buffer Bank 1
PLX PCI Local bus interface
Control/Status
Input
Input
Input multiplexer
Input
Input Triangle controller
Triangle
Tile Rendering Engine
Cont.

Figure 2. Data input for the FPGA. The input multiplexer allows the FPGA to read triangles from one bank while the PCI interface writes data to the other.

performing transfer method to the FPGA board, using the current drivers. For higher performance, the memory copy can be avoided by using chaining mode PCI bus-master DMA, but this requires an improved driver.

Once the data has arrived in one of the input buffer memory banks on the FPGA board itself, the tiles are rendered one tile at a time, using the buffer swapping input multiplexer described in [6] to allow PCI-bus data transfer and tile rendering to be overlapped in time. This buffering scheme is essentially an implementation of a very large input FIFO needed to maximize PCI bandwidth by using large transfer sizes. Figure 2 shows the input multiplexer needed to access data from one or the other bank, as well as the input triangle controller.

Note that if chaining mode DMA for transfer of smaller buffers is available, the on-chip BlockRAMs on the FPGA would be sufficient for input FIFO buffering, eliminating the need for external SRAM buffering. Incidentally the Virtex FPGA implementation currently has exactly 8 kbytes of unused on-chip BlockRAMs, just enough for two 4 kbyte input triangle buffers, usable for implementing an 8 kbyte input FIFO by utilizing the dual-ported BlockRAMs.

From this point on the FPGA implementation of the tile rendering engine from figure 1 renders the triangles in each tile, and then stores the rendered tiles in a full-frame framebuffer ready to be sent to the display. The next section discusses how the display is handled.

4. VGA VIDEO OUTPUT

One of the major bottlenecks of the FPGA implementation in [6] is the transfer of the final rendered image back to the host PC. Since this transfer is done across the same PCI-bus as the source data transfer to the FPGA, the PCI-bus is continuously switched between sending triangle data and retrieving image data to and from the FPGA board. The required bandwidth for transferring animated images is very high when high resolution images are transferred at a high framerate. This also puts the FPGA implementation of Hybris at a disadvantage compared to other 3D graphics processors which are integrated in the video display hardware. The required bandwidth for transfer of a video image sequence can be formulated as follows:

$$B_{video} = width * height * depth * framerate \tag{1}$$

where *width* and *height* specify the size of the image in pixels, *depth* is the number of bytes per pixel, and *framerate* is the number of image frames to be transferred per second. B_{video} is the bandwidth in bytes/sec. For example a low resolution image of 640x480 pixels with 8 bits/pixel at 30 frames/sec requires a bandwidth of 9.2 Mbytes/s, easily accommodated by the PCI-bus though it steals bandwidth from data transfer to the FPGA board. However the video bandwidth required for higher resolutions can be quite high. Transferring a typical image for a

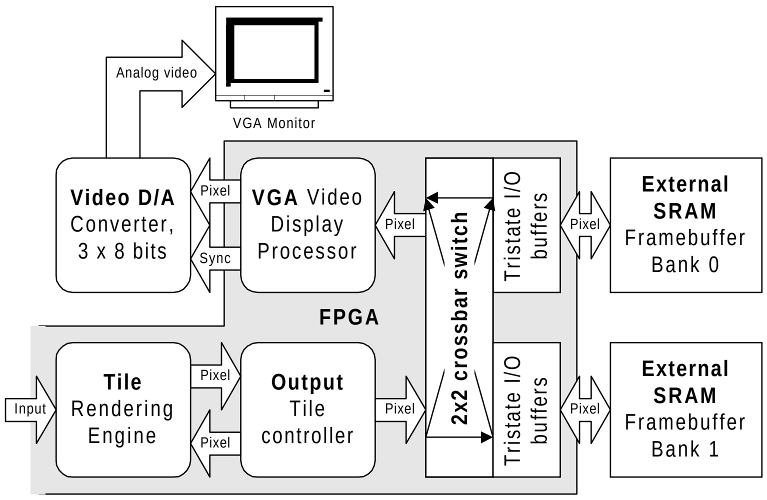

Figure 3. Integration of a VGA video output processor and the tile rendering engine in the same FPGA. A 2x2 crossbar switch and two external memory banks allows independent framebuffer access for the two processors.

PC display with a size of 1280x1024 pixel in 24 bits/pixel true color at 75 Hz full frame rate requires a bandwidth of 295 Mbytes/s, about three times higher than the available bandwidth on the PCI-bus.

The best way to get rid of this extra bandwidth requirement is to integrate the 3D graphics processor in the video display hardware to get a dedicated communication channel to the display.

Video output for a CRT (Cathode Ray Tube) based VGA monitor requires generation of five signals. Three analog colour signals for intensity of the primary colours `Red`, `Green` and `Blue`. Additionally two synchronization signals, `hsync` and `vsync` are needed for scan-line and frame synchronization. To display a digital image from a framebuffer on a CRT monitor the VGA video interface must serialize the pixel contents of the framebuffer into a continuous analog signal, and generate synchronization pulses between scanlines and frames which follows the VGA timing requirements.

In practice VGA timings are expressed using the pixel clock to determine the duration of synchronization and blanking as a number of pixel clock cycles. E.g. to accommodate a visible area of 640x480 pixels as well as the blanking periods for synchronization requires a pixel clock frequency of 25.175 MHz to allow for a 31.5 kHz line rate with a total of 800 pixels of which 25.17 μs are the 640 visible pixels and 3.77 μs (95 pixels) form the hsync pulse in the remaining blanking interval, similarly for the 60 Hz frame rate we have a total of 525 scanlines of which 480 are visible and 2 are used for the vsync pulse.

The contents of the framebuffer, which was created using the tile rendering engine, now has to be displayed on the VGA monitor. This is done by fetching one pixel from the framebuffer per pixel clock during raster-scan of the visible area. In the FPGA implementation the framebuffer is stored in double buffered external SRAM memory, allowing the tile renderer to render one frame one tile at a time to the first buffer, while a VGA display processor reads pixels in raster-scan sequence from the second buffer. The framebuffers are stored in two independent memory banks, allowing full memory bandwidth to both the rendering and video display processors. A 2x2 crossbar switch allows switching between memory banks at any time, using tri-state buffers to allow switching between input from and output to the external SRAM banks. Figure 3 shows how the video output from the tile rendering engine is handled by integrating the VGA video output processor in the same FPGA.

The current FPGA implementation uses two clock domains to allow independent operation of the tile rendering engine and the video display processor. This allows setting the pixel clock to exactly the required frequency, while the tile rendering engine's clock frequency remains fully adjustable. Since the external SRAM is asynchronous, no problems are caused by switching the SRAM banks between two clock domains. If synchronous SSRAM or SDRAM is to be used in a future implementation, a more elaborate mechanism must be used to cross between clock domains. Since the tile renderer processes one tile of 32x32 pixels at a time, the framebuffer also reflects this tiling pattern. However the video display processor requires a linear stream of pixels one scanline at a time. This is handled by reading one line of 32 pixels from each framebuffer tile intersected by the currently displayed scanline. An SRAM address calculation datapath handles this automatically. As the tiles are 32 pixels wide this framebuffer linearization method is also applicable to an SDRAM based framebuffer without too much overhead.

5. CONCLUSION

The FPGA implementation is currently very promising for future performance improvements. The implementation utilizes about half of the Virtex XCV1000's resources, and operates reliably at 25 MHz. It is able to render the 70,000 triangles Stanford Bunny at 12 frames/sec. In comparison the Nvidia Geforce2 GTS graphics processor renders the same bunny at 20 frames/sec. If we improve the internal speed of the FPGA by applying more aggressive pipelining we might run into problems with the external asynchronous SRAMs as they can be operated reliably only up to about 35 MHz from the FPGA. This was discovered during the design of the VGA video output processor, as it uses simpler logic and a different clock domain than the tile renderer, allowing its clock frequency to be set independently of the tile renderer's clock. However, since the tile renderer uses internal on-chip memory for the tile-based rendering, a future design can be allowed to run with a faster clock than the external memory. If the internal processing speed can be made far higher than the external memory speed, we may use the extra computational power to improve the rendering quality e.g. by applying anti-aliasing as discussed in [4], demonstrating the Hybris architecture's scalability. The RC1000-PP FPGA prototyping board itself has proved to be very flexible however one should be aware of its limitations. In order to implement more of the Hybris architecture with the front-end processing as well as multiple parallel tile rendering engines etc., an improved prototyping board is needed. Such a board should feature better control of the PCI bus, faster and larger external memories (e.g. DDR SDRAM) as well as a larger FPGA such as the Xilinx Virtex II or Altera APEX II.

References

[1] M. Bowen. *RC1000-PP Software User Guide*. Embedded Solutions Limited, 1998. Version 1.10.

[2] P. J. Diefenbach. *Pipeline Rendering: Interaction and Realism through Hardware-based Multi-pass Rendering*. PhD thesis, Computer and Information Science, University of Pennsylvania, 1996.

[3] T. Gleerup. ASIC for 3D Graphics Pipeline Back-End. Master's thesis, Information Technology, Technical University of Denmark, January 1999.

[4] H. Holten-Lund. *Design for Scalability in 3D Computer Graphics Architectures*. PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, July 2001.

[5] PLX Technology, `http://www.plxtech.com`. *PLX PCI 9080 Data Book*, January 2000. Version 1.06.

[6] H. A. Sørensen. SoC Design-Eksperimenter med en Stor FPGA. Master's thesis, Information Technology, Technical University of Denmark, February 2001.

[7] C. Sweeney and B. Blyth. *RC1000-PP Hardware Reference Manual*. Embedded Solutions Limited, 1998. Version 2.1.

[8] Xilinx, Inc. *Virtex™ 2.5V Field Programmable Gate Arrays*, May 2000. `http://www.xilinx.com/partinfo/ds003.htm`.