A Network Interface for Enabling Visualization with FPGAs

Craig D. Ulmer and David C. Thompson
Sandia National Laboratories*
7011 East Avenue
Livermore, California USA
{cdulmer, dcthomp}@sandia.gov

ABSTRACT

Visualization in scientific computing refers to the process of transforming data produced by a simulation into graphical representations that help scientific users interpret the results. While the back-end rendering phase of this work can be performed efficiently in graphics card hardware, the front-end "post processing" portion of visualization is currently performed entirely in software. Field-Programmable Gate Arrays (FPGAs) are an attractive option for accelerating post-processing operations because they enable users to offload computations into reconfigurable hardware.

A key challenge in utilizing FPGAs for this work is developing an infrastructure that allows FPGAs to be integrated into a distributed visualization system. We propose a networked approach, where each post-processing FPGA is equipped with specialized network interface (NI) hardware that is capable of transporting graphics commands across the network to existing rendering resources. In this paper we discuss a NI for FPGAs that is comprised of a Chromium OpenGL interface, a TCP Offload Engine, and a Gigabit Ethernet module. A prototype system has been tested for a distributed isosurfacing application.

Keywords

FPGA, visualization, networking, isosurfacing, TCP

1. INTRODUCTION

One of the challenges associated with scientific computing is interpreting the numerical results that are generated by a simulation. Simulation results can be exceptionally large data sets with subtleties that are both important to researchers and non-trivial to observe. In order to better explore these data sets, researchers often utilize visualization tools that can highlight relevant features in the data and represent regions of interest in a more insightful, graphical form. The availability of cost-effective, high-performance hardware has shaped modern visualization into a process with three distinct operations. As illustrated by the data flow of Figure 1, these phases are post processing, data staging, and graphical rendering.

1.1 Post Processing

In post processing, scientific analysis is performed on a simulation's results in order to extract information that is meaningful to the end user. Post processing can involve a variety of operations, including data transformations, statistical analysis, and integrity validation. In general, post-processing results are converted to graphical primitives that can then be rendered to a visual display. For an example of post processing, consider the case where a user wants to isolate a pressure wave as it moves in time through a three-dimensional space. Assuming that the simulation produces a three-dimensional block of integer pressure values for each time set, the user could apply an isosurfacing algorithm to locate the pressure values that exceed a particular threshold in each time step. This isosurfacing algorithm would then produce a collection of polygon primitives that approximate the surface of the threshold region. Therefore, in addition to extracting relevant features from the data set, the isosurfacing algorithm would produce data objects that could be rendered by graphics hardware.

1.2 Data Staging

Data staging in visualization refers to the process of transferring data between post processing and rendering unit(s). While trivial in a single-host system, data staging can be complex in systems such as tile display walls, where graphics objects are frequently

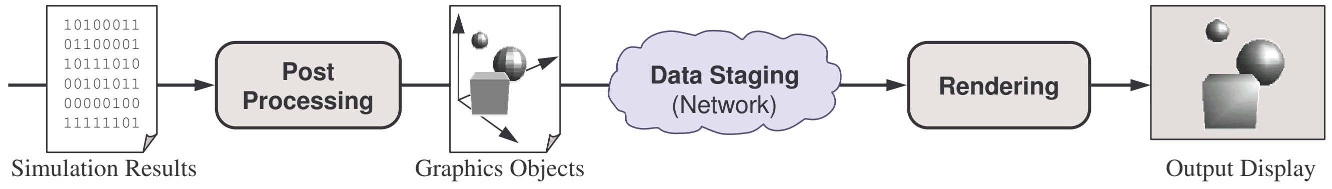

Figure 1: The process for visualizing scientific data involves post processing, data staging, and rendering.

* Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL8500.

transferred between distributed visualization resources. One popular software library for orchestrating data staging in a distributed environment is Chromium [1]. Chromium is an open-source software package for sending rendering commands over a communication link to graphics hardware. Chromium provides an Open Graphics Language (OpenGL) [2] front end that intercepts an application's graphics commands and encodes them as messages for network delivery. Chromium messages can be transferred to rendering nodes through a variety of network transports, including TCP/IP Ethernet, InfiniBand [5], and Myrinet [6]. At the rendering nodes, OpenGL commands are extracted from Chromium messages and issued to the local graphics card's device driver for rendering.

1.3 Rendering

The final phase of the visualization process is rendering. Rendering refers to the task of converting a collection of graphical objects into a displayable image. Modern systems utilize high-performance graphics cards that are capable of rapidly generating high-resolution images from data sets that contain millions of polygons. Internally these cards employ multiple graphic pipelines to convert three-dimensional polygons into two-dimensional images. Rendering is controlled through specialized graphics languages such as OpenGL [2]. While video card vendors are always expanding the capabilities and speeds of their hardware, their work is motivated by a multi-billion dollar gaming and entertainment market. Unfortunately, the technical needs for this market are generally different than those found in scientific computing. Therefore it is unlikely that commercial graphics cards will directly support the post-processing operations required by the scientific computing community in the near future.

1.4 Modern FPGAs

Field-Programmable Gate Arrays (FPGAs) are reconfigurable hardware devices that can be programmed to emulate large, digital hardware circuits. Over the last decade researchers have used FPGAs as computational accelerators in a variety of applications. Recent advances in commercial FPGA architectures have renewed interest in this form of reconfigurable computing. Modern "platform" FPGAs feature large amounts of reconfigurable logic as well as special-purpose hardware units that can be leveraged by designers. For example, the Xilinx Virtex-II/Pro (V2P) FPGA architecture [3] includes reconfigurable logic, one or more PowerPC processors, internal memory, and flexible transceivers that can interact with high-speed networks such as Gigabit Ethernet [4], InfiniBand [5], and Myrinet [6]. These components enable researchers to integrate networking capabilities into their FPGA work, and therefore allow FPGAs to be utilized in new manners.

2. VISUALIZATION WITH FPGAS

The scientific community's continuous thirst for powerful visualization systems motivates us to consider architectures where post processing can be performed more efficiently. FPGAs are an attractive technology for this work because FPGAs enable users to cost-effectively implement key portions of an algorithm in fast, custom hardware. Adapting post-processing algorithms to function in hardware instead of software can result in significant performance improvements that increase the overall quality and usability of a visualization application.

However, before FPGAs can be utilized in this regard, an important architectural question must be addressed: how should FPGAs be integrated into visualization systems that already employ powerful rendering resources?

While there are multiple strategies for addressing this challenge, most have negative side effects. For example, the most straightforward approach would simply be to equip a workstation with an FPGA card for post processing and a video card for rendering, and then instruct the host processor to move data between resources as needed by the application. Unfortunately, this approach suffers from scalability issues because processing is limited to the resources that are available in the local host. At the other end of the spectrum, integration could be addressed by ignoring existing rendering resources and implementing both post-processing and rendering operations in the FPGA. In addition to duplicating industry efforts, this approach is unlike to provide competitive results compared to a system that uses commodity video card hardware for rendering. What is needed is an integration solution that can (1) leverage existing rendering hardware/software and (2) be scaled to an architecture that supports hundreds of post-processing and rendering resources.

2.1 A Networked Approach

In this paper we present an alternative strategy for incorporating post-processing FPGAs into the distributed visualization environment. We propose a networked approach, where each post-processing FPGA is loaded with a specialized network interface (NI) circuit that is capable of transmitting graphics commands over the network to a remote host for rendering.

While a NI consumes FPGA resources and is nontrivial to implement, this approach is beneficial for multiple reasons. First, the networked approach decouples post processing hardware from rendering hardware. This trait enables us to focus on developing post-processing accelerators in FPGAs while leveraging existing graphics cards for rendering. Second, this approach is scalable because the communication network functions as the fabric for interconnecting our visualization resources. As such, the system can easily be expanded by (1) attaching additional FPGAs or rendering nodes to the network and (2) increasing the network's routing resources to meet bandwidth requirements. Finally, if the NI is designed to work with existing standards for transporting graphics commands, post-processing FPGAs can be connected to current visualization system that are already in place. This feature therefore leverages existing hardware/software investments and does not require radical changes in current practices.

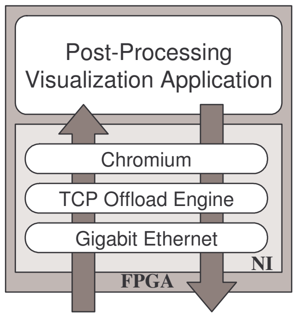

Figure 2: A Network Interface for FPGA-based Visualization

In order to enable post-processing visualization research with FPGAs, we have implemented a NI that is capable of transporting graphics commands to a remote host for rendering. This NI is written in the Verilog hardware description language [7], and currently targets the Xilinx Virtex II/Pro (V2P) FPGA architecture. As depicted in Figure 2, this NI is comprised of three components: a Gigabit Ethernet (GigE) module for managing low-level network interactions, a TCP Offload Engine (TOE) for reliable communication, and a Chromium module for transporting OpenGL graphics commands.

The remainder of this paper describes our experiences in designing and working with this NI. Sections 3 and 4 provide details about individual NI components and their performance. Section 5 presents an example of an FPGA-based, post-processing application. A functional prototype of the application with the NI is then presented in Section 6 to demonstrate a complete, working system. Finally, the paper is concluded with observations about this work and a discussion of opportunities for future research.

2.2 Experiment Environment

Due to the nature of this research effort, it was necessary to create an experiment environment that enables us to observe how our implementations behave when connected to commodity network hardware. For this work we utilize a commercial stand-alone FPGA card that is connected to a PC through a GigE link. The FPGA card is an Avnet Virtex-II/Pro Development Kit [8] from Avnet Design Services. This development board is equipped with a Xilinx V2P20 FPGA and two small-form pluggable (SFP) receptacles that are loaded with GigE optical transceivers. The host PC has dual 2.6 GHz Xeon processors, an NVIDIA Quadro4 AGP video card, and an Intel 8245EM GigE controller. The PC is loaded with the Linux 2.4.20 operating system and the December 2004 version of the Chromium software library. Additional FPGA network compatibility studies have been conducted using a Packet Engine G-NIC network interface card. Hardware designs are compiled using the Xilinx ISE 6.3 design tool chain, which performs synthesis through the Xilinx Synthesis Tool (XST).

2.3 Related Work

There have been a number of relevant FPGA-related research projects over the last decade that have influenced our work. In terms of visualization, several researchers have reported on the use of FPGAs in graphics application. A complete FPGA-based volumetric rendering system is presented in Vizard-II [9]. A real-time ray tracing architecture for FPGAs was developed in the SaarCOR project [10]. In terms of traditional polygon rendering efforts, the Sepia [11] project used FPGAs to composite data extracted from multiple video cards in a distributed system to facilitate real-time rendering. This work used external ServerNet network hardware for communication between FPGAs.

Researchers have also reported on their experiences with connecting FPGAs to communication networks. In various research efforts, FPGAs have been connected to ATM [12], Gigabit Ethernet [13], and Myrinet [14, 15]. However, all of these efforts utilized external network interface hardware to facilitate the communication. There have been relatively few academic papers that document experiences with the high-speed transceivers found in recent FPGAs, with the exception of network security applications [16]. Perhaps the greatest source of information for FPGA-based networking can be found in design documents from the FPGA vendors. For example, Xilinx provides a reference design that implements a Gigabit Ethernet NI in FPGA logic, and connects it to an on-chip processor that runs TCP in software [17].

3. NI COMMUNICATION LAYER

Our networked approach to integrating FPGAs into visualization systems is based on the design of a special-purpose NI for FPGAs. This NI provides two layers of functionality for supporting distributed visualization: reliable network communication and graphics primitive transport. In the layer that performs reliable network communication, NI hardware interacts with the network fabric and guarantees that data is properly transmitted between the post-processing application and the rendering node. For this task, we focus on an approach that utilizes FPGA hardware to implement TCP over a GigE network fabric.

3.1 External vs. Internal NI Hardware

Prior to the current generation of platform FPGAs, the only means by which an FPGA could be connected to a local area network was through the use of an external NI chip. As such, FPGA researchers with network applications have historically either built custom FPGA boards that employ external NI chips, or connected add-on network cards to an FPGA board through standard I/O interfaces such as the PCI Mezzanine Connector (PMC). In either case, designers were faced with board-level design issues as well as the task of incorporating circuitry in the FPGA to interact with the external NI.

Platform FPGAs provide an attractive alternative for connecting FPGAs to the network because these FPGAs feature on-chip transceivers that can interact directly with a network at the physical layer. These transceivers therefore present an opportunity for implementing a complete NI inside the FPGA without external circuitry. For our research purposes, there are two distinct advantages for integrating the NI into the FPGA. First, integration reduces the need for external circuitry and therefore makes a design more portable. As evidence, we cite that our NI design work was easily ported from one commercial FPGA board to another, simply by updating the top-level pin outs for the design. Second, integration enables customization in the design because the NI is implemented in FPGA logic. This trait provides a great deal of freedom to tailor the NI to both the application and the network fabric.

3.2 An On-Chip TCP/IP NI

Developing a reliable, network communication engine for FPGAs is dependent on the network substrate selected for the visualization system. For our research environment we have selected a network fabric that utilizes Gigabit Ethernet (GigE) and the Transmission Control Protocol (TCP) [18]. While impractical for high-performance computing, TCP on GigE is well understood and widely deployed. Based on the availability of platform FPGAs with built-in transceivers, we have also decided to implement the communication layer for the NI entirely in FPGA logic. The overall architecture of this layer is presented in Figure 3, with separate modules for GigE and TCP Offload Engine modules.

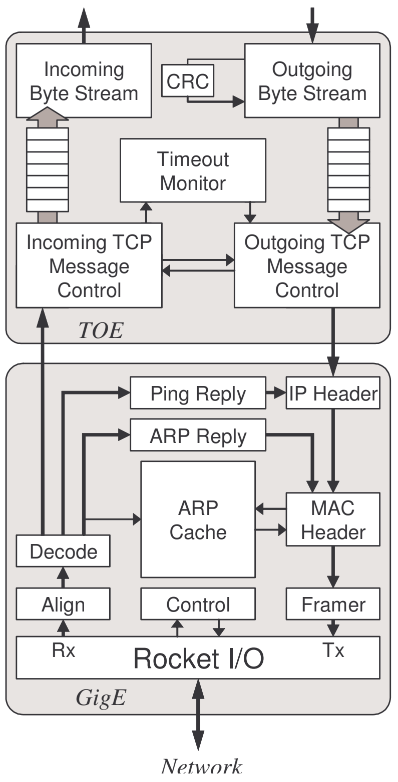

Figure 3: The NI communication layer is comprised of Gigabit Ethernet (GigE) and TCP Offload Engine (TOE) modules.

3.3 Gigabit Ethernet (GigE)

Low-level interactions with the network are handled through a GigE module. The components in this module perform the following functions.

- **Rocket I/O Transceiver:** A Xilinx V2P Rocket I/O transceiver is utilized for physical layer interactions with the GigE medium. The transceiver performs serialization/deserialization (SERDES), 8B/10B encoding/decoding, embedded clock recovery, and CRC generation/validation. GigE's 1.25Gb/s serial data rate is generated using either a 62.5 MHz or 125 MHz low-skew reference clock and the transceiver's 20x or 10x clock multiplier. User circuitry supplies packet data at a 1.0 Gb/s data rate through a 16-bit, 62.5 MHz data bus. Additional logic monitors the transceiver's state and invokes maintenance signaling when necessary.

- **Framing:** The GigE module provides units for creating and parsing protocol information at the Ethernet frame, MAC, and IP levels. These interfaces simplify the work that must be done by external modules for generating and consuming valid packets.

- **Addressing:** IP-to-Ethernet address translation is performed in the GigE module automatically using a 256-entry, direct-mapped address translation cache. This cache is updated when address resolution protocol (ARP) messages are received from the network. The GigE module consults the cache when building an outgoing message. If a translation is not available, the GigE module automatically stalls the message and transmits an ARP request message to locate the unknown IP address.

- **PING Handling:** The GigE module provides optional support for responding to ICMP PING messages. In addition to serving as a means of detecting whether the FPGA is networked and functioning, the PING circuitry can be used as a simple mechanism for triggering application operations. For example in several of our experiments, we designed our FPGA applications to wait unit a PING message is received before attempting to open a network connection. This approach allows us to use built-in network operations as a means of controlling the experiment.

3.4 TCP Offload Engine (TOE)

For lossy networks such as Ethernet, reliable transmission protocols can be employed in order to guarantee that application data is transferred properly between sender and receiver. While complex, the transmission control protocol (TCP) provides this functionality and is available for nearly every networked system in use today. We have constructed a TCP Offload Engine (TOE) for FPGAs that is designed to manage a single TCP network connection. It is comprised of four types of units:

- **Protocol Management:** At the heart of the TOE module is a pair of protocol engines that maintain the TCP connection. The incoming TCP message control unit parses incoming packets and extracts both state updates and appends to the user's data stream. The outgoing TCP message control unit generates control messages as well as data transmissions. Due to the limited buffer space in the FPGA, the state machines employ a "go-back-n" retransmission policy rather than "selective repeat".

- **Timeout Monitor:** The TOE employs a timeout monitor to trigger the outgoing TCP message control unit to invoke retransmission mechanisms when a response is not detected within a specified amount of time. These mechanisms reset the outgoing FIFO back to the oldest unacknowledged message, and then retransmit all messages in the queue.

- **Packet FIFOs:** The TOE utilizes a pair of packet FIFOs to convert between an application's byte stream interface and TCP packets. FIFOs are 32-bits wide and can store 8 KB of data each (configurable at build time). The packet FIFOs are designed to allow rollback (e.g., if an incoming TCP message is about to saturate the buffer and needs to be dropped, or if the outgoing engine needs to rollback to a particular point in order to retransmit a message).

- **TCP CRC Generation:** One of the hardships of TCP is that the TCP message checksum is stored in the header of the message instead of the tail. In order to remove the need for scanning a message more than once, a checksum engine is built into the outgoing message FIFO user interface. This unit calculates a partial CRC for each message as it is written into the FIFO. The outgoing TCP message control unit uses this information to build the full checksum when the message is transmitted.

3.5 Performance

The TOE/GigE portion of the NI was implemented and adapted for use with the ADS FPGA board. A series of experiments were then conducted to observe the communication performance of the FPGA with a host PC. A test module was constructed for the FPGA that uses the NI to transmit several bursts of data through the TCP connection to a host application. The host application extracts the data from a standard TCP socket and measures the amount of time required to send a series of bursts. The connection is warmed prior to any measurements to remove TCP slow-start effects.

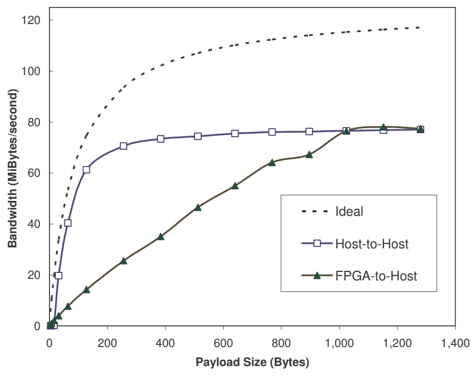

Figure 4: Measured TCP bandwidths for FPGA-to-Host and Host-to-Host transmissions.

The results of the bandwidth test are presented in Figure 4, along with ideal values for a TCP connection and values measured for data transfers between a pair of hosts. From these tests we see that FPGA-to-Host bandwidth increases at roughly a linear rate with packet size, and that Host-to-Host bandwidth saturates much earlier. This difference can be attributed to the fact that the FPGA TOE does not implement the Nagle algorithm [19] which combines a series of small bursts into a larger packet. In any case, the FPGA implementation provides comparable performance for larger packet sizes. Given that visualization applications generally transport large blocks of data, this performance is acceptable for our applications.

3.6 Observations

The GigE/TOE combination provides a basic communication engine that enables us to reliably transmit data between an FPGA and a host computer using a standard GigE network. While our initial approach in this effort was to implement a only subset of TCP/IP functionality, we discovered that it was nearly impossible to build a working system that did not handle all of the subtle behaviors of TCP/IP. Over the course of development, the TOE grew to include support for slow start, NACK detection, and rate throttling. Additional work for combining small messages (i.e., the Nagle algorithm [19]) has been implemented, but is not discussed in this paper due to the large packet sizes used by our applications. The functional requirements for the GigE module expanded in a similar manner in order to maintain interoperability with commodity network hardware. Implementations that lacked these enhancements did not function well, if at all, with host computers that were equipped with commercial network components.

From a user's perspective, the TOE/GigE communication engine is appealing for multiple reasons. First, the TOE provides a simple byte-stream API that is easy to use. The fact that the TOE handles reliable transmissions over the network simplifies the amount of work that higher-level modules (e.g., Chromium) must perform. Second, the TOE/GigE communication core is self-contained and easily replicated. This trait enables designers to easily instantiate multiple NIs on each FPGA as resources permit. Finally, because the communication module's API is not TCP/IP specific, it is possible for users to replace the module with hardware that utilizes different network substrates (e.g., InfiniBand) or protocols (e.g., reliable UDP or a more full-featured TCP). This flexibility ensures that applications can be written in a manner that is indifferent to the underlying network technologies.

4. NI GRAPHICS TRANSPORT LAYER

The second layer of the FPGA NI for visualization is responsible for graphics transport. This layer translates a post-processing application's graphics operations into commands that can be transported across the network to a host for rendering. While it is possible to implement this functionality in a variety of manners, we advocate an approach that leverages existing standards. In particular, we propose utilizing the OpenGL standard as the user's programming interface and Chromium as the interface for transporting OpenGL commands over the network. For simplicity, we refer to the graphics transport hardware implemented in the NI as the Chromium module.

The Chromium module utilizes a reliable network transport module such as the TOE to establish and maintain a connection with the remote host that is responsible for rendering the applications graphics commands. Once the TOE establishes a connection, the Chromium module exchanges a small amount of information with the remote rendering application to share context information. After initialization, the FPGA application is free to issue OpenGL commands. The Chromium module packages a sequential list of OpenGL commands into a standard Chromium packet that can be transported across the connection. While the current implementation of the Chromium module only implements a basic subset of the OpenGL commands that Chromium is capable of transporting, new commands can be added simply by updating a translation table that associates a particular OpenGL command with a Chromium-specific identifier.

4.1 Chromium Messages

Chromium is designed to pack data as efficiently as possible into a maximum transfer unit (MTU) of the underlying network substrate. Packets are comprised of three sections: a header, a list of commands, and a data payload section. The 16-byte header for the message contains basic information for the message, including the identity of the OpenGL context that the message is destined for. The commands section of a message holds one or more 8-bit opcodes that correspond to the specific OpenGL actions that are to be invoked at the rendering node. Opcodes are packed in reverse order and then zero padded to align the section on a 32-bit boundary. Each opcode has one or more 32-bit data values that are stored sequentially in the data section of the message.

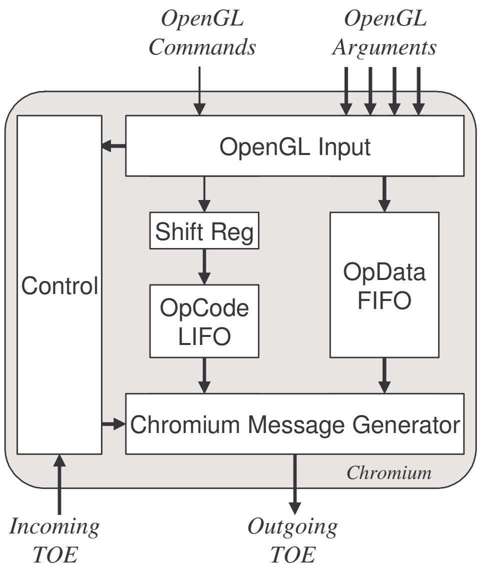

Figure 5: The architecture of the Chromium Module.

The architecture for the Chromium module is illustrated in Figure 5. Internally the Chromium module utilizes two blocks of memory for packet assembly: one for opcodes and the other for opcode data. Opcodes are packed as they arrive using a shift register and stored in a LIFO to reverse the opcode sequence. The data section is assembled in a FIFO. As the FIFO and LIFO approach capacity, a high-water flag is asserted to notify the visualization application that only a few more operations can be inserted before the message needs to be flushed. This approach enables the application to gracefully terminate a stream of OpenGL commands. Once the application issues a flush command, the Chromium module assembles the message and delivers it to the TOE. In cases where the TOE does not have appropriate buffer space, the Chromium engine blocks until the data can be accepted.

4.2 Interface Implementations

Three versions of the Chromium module were constructed in order to implement different APIs. The first two implementations utilize a blocking interface that requires explicit handshaking whenever an application inserts a new OpenGL command into the data stream. The blocking modules employ either a single-buffered of double-buffered approach to assembling Chromium packets. The third implementation of the Chromium module implements a non-blocking interface where user applications can stream commands into the module without handshaking overhead until the buffers reach capacity. The non-blocking implementation is double buffered. Simulation and synthesis experiments were performed to observe the tradeoffs involved in these interfaces. For the simulation work, a design was constructed to measure the number of clock cycles required to convert 1,000 OpenGL triangles into the corresponding Chromium packets. Results from these experiments are presented in Table 1.

Table 1: Implementation details for the Chromium module.

Interface	Blocking		Non-Blocking
Buffering	Single	Double	Double
Slices	757	772	814
BRAMs	2	6	12
Max Clock Frequency	186 MHz	200 MHz	85 MHz
Clocks for 1K Triangles	29,537	15,450	12,824

It is necessary to examine both the synthesis and simulation results to select the most appropriate Chromium module for a design. Simulation experiments confirmed that the non-blocking, double-buffered implementation required the least number of clock cycles to process 1,000 triangles due to reduced signaling overhead. However, as the synthesis results reveal this performance comes at the cost of additional hardware resources and a much lower maximum clock frequency. Overall, the blocking double-buffered approach provides the best tradeoff in terms of performance and resources, and is therefore the recommended implementation choice. However, the other implementations are available for design situations that have tighter or looser resource requirements.

4.3 Performance Measurements

A series of experiments were performed using the ADS FPGA board and a host PC to observe the communication performance of the visualization NI. For these tests we constructed a triangle generation module that supplies the NI with a stream of OpenGL commands for drawing triangles. Three different host applications were utilized to measure data rates at various points in the rendering system. The first program (Network) simply extracts all incoming data from the TCP socket and discards it. This program provides an estimate of the raw rate at which the NI can stream triangle data over a connection. The second program (Rendering NOP) uses the Chromium software library to parse incoming messages, but does not render the results. This program provides an estimate of parsing overhead. Finally, the last program (Rendering Full) uses the Chromium library to parse messages and render the results to the display. This program provides an end-to-end performance measurement for the system. Performance is measured in terms of thousands of triangles transferred per second. A single triangle is comprised of 49 bytes of data.

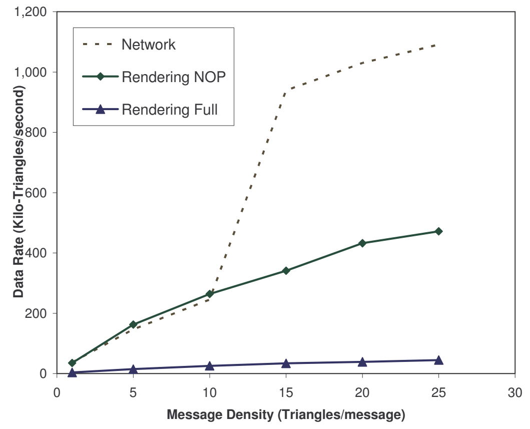

Figure 6: Measured performance for transporting triangles from FPGA to Host, based on action performed by host.

The results of the experiments are presented in Figure 6. From these results we observe that there is a considerable amount of overhead associated with the Chromium software library. The raw TCP connection was able to supply a million triangles per second (or approximately 400 Mb/s of triangle data). Using Chromium to parse these messages dropped performance to 450 thousand triangles per second (or 168 Mb/s of triangle data). A complete system with rendering dropped performance to 50 thousand triangles per second (or 20 Mb/s of triangle data).

5. AN ISOSURFACING EXAMPLE

In order to demonstrate the functionality of the visualization NI, we have constructed a post-processing application for FPGAs that performs isosurfacing on a three-dimensional data set. Isosurfacing in this context refers to the process of thresholding a multidimensional data set in order to locate features that are of interest to the user. For example, isosurfacing is often utilized in medical imaging applications. These applications use isosurfacing to locate bone or tissue structures within a volume of data generated by computed tomography (CT) or magnetic resonance imaging (MRI) equipment.

Our isosurfacing implementation is based on the Marching Cubes [20] algorithm. This algorithm decomposes a volume of input data into a collection of data cubes and then analyzes each cube individually. Each of the eight values in a cube is compared against a threshold value to determine which of the cube's edges intersect the isosurface. A lookup table is then used to transform this information into a list of triangles that best approximate the intersection surface within the cube. Our implementation employs integer operations for its calculations and uses edge midpoint values to approximate intersection points.

5.1 Isosurfacing Module Data Flow

The architecture of the isosurfacer module constructed for this effort is depicted in Figure 7. The three-dimensional volume of 32-bit input data values is stored in an external block of memory. The fetch unit retrieves four input values at a time from the memory and repeats the process three more times until a stack of three data cubes is assembled for processing. Data cubes are then individually streamed into the module's analysis units. During analysis, each value in the cube is compared to a threshold value to create an 8-bit signature for the cube. Concurrently, midpoint coordinates for all the edges in the cube are computed. If the threshold signature is all ones or all zeros, the cube does not intersect the threshold and can be dropped. Otherwise all data for the cube is stored in a slack buffer for later processing. The final unit in the module translates a cube's threshold signature into a collection of triangles that approximate the surface. This translation is facilitated by a 256-entry table that specifies the (2) number of triangles to be generated and (2) the edge midpoints to use for each triangle vertex.

The dataflow for the isosurfacing module is designed to maximize processing performance while handling output generation rates that vary based on the input data set. The fetch unit obtains four

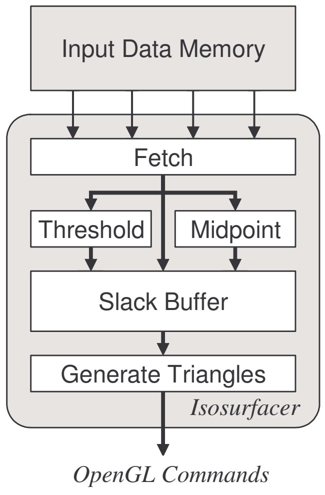

Figure 7: Architecture of the isosurfacing module.

column values at a time in order to match the rate at which cubes are analyzed. The slack buffer is extremely wide (488-bits) and can accept a new cube of data every clock cycle until the buffer is saturated (1,024 entries). This buffer effectively decouples the front-end analysis work from the back-end triangle generation, and is necessary because each cube (eight 32-bit values) can generate up to five triangles (each with three 3x32-bit vertices).

5.2 Simulation Example

Prior to synthesis, the isosurfacing module was tested using the ModelSim Verilog simulator and a test bench design. The test bench instantiates a large block of memory for housing input data, an isosurfacing module, and a unit for writing output vertex data to a file. This data is then viewed offline using an external C program. Results generated by the hardware simulation are also validated against those generated by a software implementation of the isosurfacing algorithm.

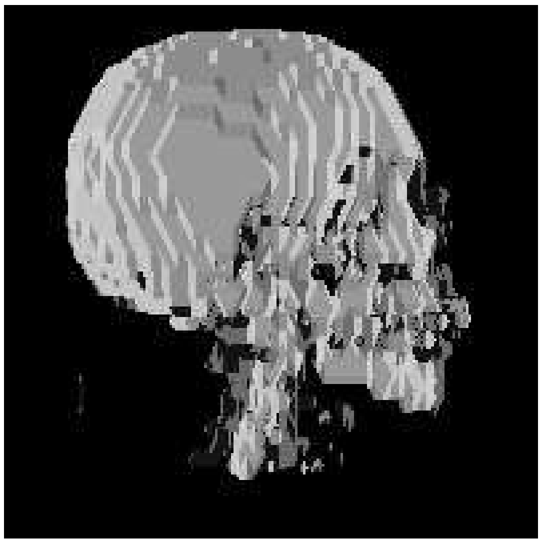
Figure 8: Hardware-based isosurfacing of 64^3 CT data set.

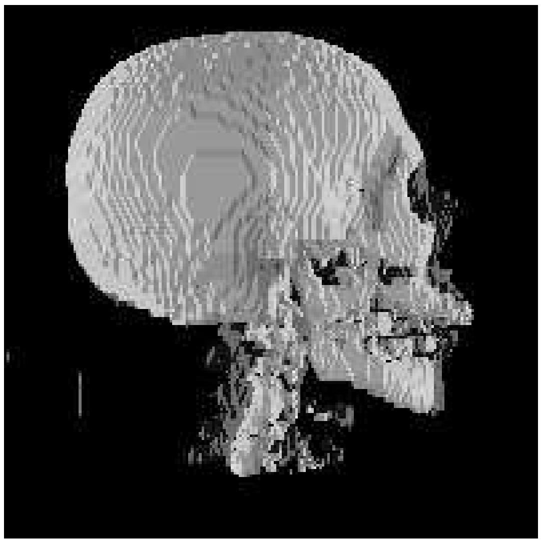
Figure 9: Software-based isosurfacing of 128^3 CT data set.

In order to verify the functionality of the hardware, the memory unit in the simulation test bench was loaded with a 64^3 data set of CT values. This data is courtesy of the North Carolina Memorial Hospital and was acquired with a General Electric CT scanner [21]. A rendering of the 30,391 triangles produced by the hardware simulation is presented in Figure 8. While this rendering provides a rough approximation of the patient's skull, the surface is blocky due to (1) the low resolution of the data set and (2) the implementation's use of integer arithmetic for calculating edge intersections.

For comparison purposes, the software implementation of the algorithm was utilized to process a 128^3 value version of the data set. The software produced the 138,428 triangle rendering presented in Figure 9. From this image it is clear that increasing the input resolution enhances the overall quality of the image. However, the integer calculations used in this algorithm still result in blocky features in the image. These flaws can be obscured through standard shading techniques, but are omitted from this work in order to provide a detailed view of the algorithm's raw output.

6. A FUNCTIONAL PROTOTYPE

Our vision for this research is one where a host PC is equipped with an FPGA board that offloads post-processing operations and transmits results over the network to a rendering node. As a means of demonstrating this vision, we have constructed a functional prototype that incorporates an isosurfacing module and a visualization NI into the ADS board's FPGA. Due to a lack of support with this board for PCI interactions, we decided to replace the host PC's CPU in our model with one of the V2P's on-chip PowerPC CPUs. While this substitution ignores intra-PC I/O effects, it provides a flexible platform for experiments where a software application can utilize a hardware unit for offloading post processing operations.

As illustrated in Figure 10, the prototype design features a visualization NI, an isosurfacing module, and a PowerPC CPU for hosting software applications. The PowerPC is connected to two blocks of on-chip memory. The first houses the application's executable data and is attached to the CPU through a Processor Local Bus (PLB) that is implemented in FPGA logic. The second memory block houses input data for the isosurfacing module, and is connected to the PowerPC through its data-side on-chip memory controller (DSOCM). Additional logic is included in this interface to implement memory-mapped control/status registers that enable the PowerPC to control the isosurfacing module.

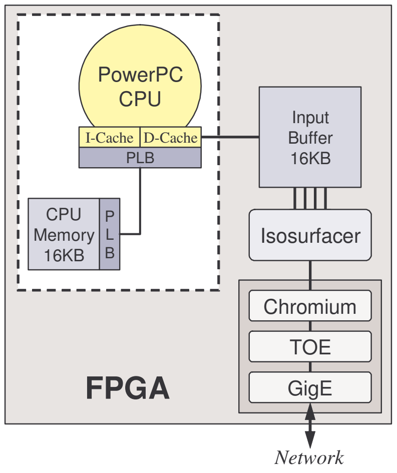

Figure 10: The post-processing system FPGA prototype.

6.1 Resource Utilization

The prototype design was compiled for the ADS board's V2P20 FPGA and examined to determine resource utilization for the hardware. Overall the design consumed 79% of the V2P20's available logic and 86% of its on-chip memory. Table 2 presents estimates for both logic (i.e., FPGA slices) and memory (i.e., FPGA Block RAMs) utilization for each of the modules in the design. The units requiring the most resources are the isosurfacing module and the TOE. The isosurfacing module consumes significant memory resources because it employs a wide data path and deep buffers. The TOE's resource requirements are due to its complex protocol processing state machines and its use of packet FIFOs (16 KB/6KB for outgoing/incoming data).

Table 2: Resource utilization estimates for the prototype.

Module	Slices (% of V2P20)	Block RAMs (% of V2P20)
PowerPC	639 (7%)	8 (9%)
Isosurfacing	1,286 (14%)	48 (55%)
Chromium	801 (9%)	6 (7%)
TOE	2,990 (32%)	11 (12%)
GigE	1,592 (17%)	3 (3%)

6.2 Verification

The hardware portion of the prototype design was compiled and loaded on the ADS board's FPGA. We then constructed a software application for the PowerPC that generates a sequence of data values that are written into the input buffer of the isosurfacing module. While the 16 KB capacity of the input buffer limits isosurfacing data sets to 16^3 values, the resolution was suitable for verifying the system worked properly with simple objects. The prototype was connected through a GigE link to a host PC and then instructed to begin processing data. A Chromium application on the PC captured the FPGAs data and rendered the results to a display. We verified that the spherical data set generated by the PowerPC matched the output rendered by the Chromium application. This experiment confirmed not only that the system operated properly, but also that the FPGA could be integrated into an existing visualization system through the networked approach.

7. OBSERVATIONS

The broad scope of this project has provided us with a significant amount of insight into multiple topics of FPGA research. From a networking perspective, modern platform FPGAs are now capable of interacting with commodity network hardware through their flexible, on-chip transceivers. However, it is important to note that designers must implement a moderate amount of control logic in the FPGA in order to ensure that the FPGA interacts properly with the network. The new Virtex-4 FX [22] FPGA architecture simplifies this development because it provides built-in MAC units for GigE. However, other networks such as InfiniBand still require NI implementations similar to the GigE module presented in this paper.

Our TOE implementation demonstrates that a self-contained TCP unit can be implemented in hardware for current generation FPGAs. While the TOE consumes 32% of a V2P20's logic, this FPGA is one of the smaller parts available in the V2P family. From a development perspective, constructing the TOE was nontrivial and time consuming due to the subtleties of TCP behavior. In retrospect we observe that a simpler approach would have been to instantiate a soft processor in FPGA logic and port a software version of TCP to the processor. However, the TOE implementation is built and will be released as open source hardware for the public.

Our experiences with Chromium indicate that it is a suitable transport layer for graphic commands. The OpenGL API is well understood by visualization researchers and Chromium utilizes a well-packed data format for transmissions. While this format is easy to assemble on systems that employ random access memory, we note that it does not map well to a streaming architecture. As a result, the Chromium module must capture an entire message in LIFO/FIFO elements before any portion of it can be transmitted.

The isosurfacing application provided a real-world example of post processing that is commonly used in current visualization work. The implementation confirmed that the FPGA could support very wide data paths, as well as application-specific pipelines. Additionally, the design's slack buffer enables the fast, front-end analysis components to operate independently of the slow, back-end triangle generators, until saturation occurs. These types of optimizations are critical for achieving high performance in hardware designs.

7.1 Future Work

There are a number of opportunities for future work in this field. In terms of performance, we note that the primary bottleneck in the prototype system is the network substrate. We have found that the Chromium modules are capable of producing data streams at near GByte/s data rates. Therefore, the next logical step in this work would be to move to a faster network substrate such as InfiniBand. In addition to providing higher-bandwidth, InfiniBand provides support for attaching peripheral devices to the network and is therefore more appropriate for distributed visualization research.

The isosurfacing example presented in this paper suffers from two significant limitations that preclude its use in real applications. First, isosurfacing applications require significant amounts of memory in order to work with high-resolution data sets. For example, 64 MB is required to house a 256^3 data set of 32-bit data values. Therefore, future work would replace the on-chip memory interface with a controller for external (DDR) memory. Second, in order to provide high-quality images, it is necessary to update the design to use floating point operations and calculate the actual edge intersection points instead of approximating them with edge midpoint values. Our initial experiments with single-precision floating-point cores indicate that these units incur enough overhead to warrant adjusting the module's data flow.

Finally, this work would benefit from an adaptation to other FPGA accelerator boards. An ideal FPGA accelerator board would provide a high-bandwidth connection to the host (e.g., PCI-Express), a large FPGA (e.g., a V2P50 or higher), a large block of DDR memory, and multiple network ports that are driven by the FPGA's Rocket I/O transceivers. This architecture enables researchers to construct a network interface card that performs application-specific processing on messages as they are exchanged with the network.

8. CONCLUDING REMARKS

FPGAs are and attractive option for visualization research because they enable algorithms that are currently implemented in software to be adapted to efficient, custom-built hardware. Integrating FPGAs into a functional visualization system can be achieved by adding a special-purpose network interface to the FPGA that facilitates the transport of graphics data over a local area network. In this paper we have demonstrated such a NI, and have used it to send graphics primitives over the network to a host for rendering. By equipping the NI with a standard API for transporting graphics commands, we are able to leverage existing hardware and software for rendering. End-to-end tests of this hardware revealed that the NI implementation gave reasonable performance for the network substrate that was utilized.

We constructed an isosurfacing application for this work to demonstrate our vision of how post-processing operations can be performed in FPGA hardware. While the current implementation is limited in terms of input resolution and data precision, it illustrates our vision of a distributed system with a functional prototype.

References

[1] G. Humphereys, M. Houston, R. NG, R. Frank, S. Ahem, P. Kilchner, J. Klosowski. "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters," in SIGGRAPH 2002.

[2] M. Segal, K. Ashley, " The OpenGL Graphics System: A Specification," 2004

[3] Xilinx Inc. "Virtex-II Pro and Virtex-II Pro X Platform FPGAs Data Sheet," 2005.

[4] IEEE Standard 802.3z 1998.

[5] The InfiniBand Trade Organization, "InfiniBand Architecture Specification," 2002.

[6] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su, "Myrinet: A Gigabit-per-Second Local-Area Network," in IEEE Micro Vol. 15 No. 1, 1995.

[7] IEEE Standard p1364-2001, "IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language," 2001.

[8] Avnet Design Services whitepaper, "Xilinx Virtex-II Pro Development Kit," 2003.

[9] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, R. Proksa, "VIZARD II: a reconfigurable interactive volume rendering system," in Proceedings of SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002.

[10] J. Schmittler, S. Woop, D. Wagner, W. Paul, and P. Slusallek, "Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip," In SIGGRAPH/EUROGRAPHICS Graphics Hardware 2004.

[11] L. Moll, A. Heirich, and M. Shand, "Sepia: scalable 3D compositing using PCI Pamette," in IEEE Symposium on Field-Programmable Custom Computing Machines, 1999.

[12] J. Lockwood, N. Naufel, J. Turner, D. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in Proceedings of the ACM International Symposium on Field Programmable Gate Arrays, 2001.

[13] K. Underwood, R. Sass, W. Ligon III, "Cost Effectiveness of an Adaptable Computing Cluster," in the Proceedings of the 2001 ACM/IEEE Conference on High Performance Networking and Computing, 2001.

[14] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, 1999.

[15] C. Ulmer, C. Wood, S. Yalamanchili, "Active SANs: Hardware Support for Integrating Computation and Communication," in Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA, 2002.

[16] C. Clark and C. Ulmer, "Network Intrusion Detection Systems on FPGAs with On-Chip Network Interfaces," in Proceedings of the International Workshop on Applied Reconfigurable Computing, 2005.

[17] Xilinx Inc., "Application Note 536 (XAPP 536): Gigabit System Reference Design." June 3, 2004.

[18] Request for Comments RFC793. "Transmission Control Protocol", 1981.

[19] Request For Comments RFC896. "Congestion Control in IP/TCP Internetworks," 1984.

[20] W. Lorensen and H. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", in Proceedings of SIGGRAPH 1987.

[21] Stanford Volume Data Archive, http://graphics.stanford.edu/data/voldata

[22] Xilinx, Inc. Datasheet. "Virtex-4 Family Overview," 2005.