RIFFA 2.0: A REUSABLE INTEGRATION FRAMEWORK FOR FPGA ACCELERATORS

Matthew Jacobsen, Ryan Kastner

Computer Science and Engineering
University of California, San Diego
La Jolla, CA USA
{mdjacobs, kastner}@cs.ucsd.edu

ABSTRACT

We present RIFFA 2.0, a reusable integration framework for FPGA accelerators. RIFFA 2.0 provides communication and synchronization for FPGA accelerated applications using simple interfaces for hardware and software. Our goal is to expand the use of FPGAs as an acceleration platform by releasing, as open source, a framework that easily integrates software running on commodity CPUs with FPGA cores. RIFFA 2.0 uses PCIe to connect FPGAs to a CPU's system bus. RIFFA 2.0 extends the original RIFFA project by supporting more classes of Xilinx FPGAs, multiple FPGAs in a system, more PCIe link configurations, higher bandwidth, and Linux and Windows operating systems. This release also supports C/C++, Java, and Python bindings. Tests show that data transfers between hardware and software can saturate the PCIe link to achieve the highest bandwidth possible.

1. INTRODUCTION

FPGAs and GPUs have become popular parallel computing platforms for application acceleration. Both have been successfully applied to accelerate numerous vision [1], physics [2], and other compute intensive applications [3]. They are even used in heterogenous computing environments for high performance computing [4]. Both hardware devices are capable of running highly parallel operations faster than their CPU counterparts. However, many differences exist between the hardware platforms. The focus of this paper is on the difference we feel is most critical to FPGAs continued success in application acceleration; the ability for FPGAs to easily integrate with the CPU workstation environment.

Workstation CPUs are still the dominate platform for most compute intensive applications. They are easy to program, offer considerable memory, many processing cores, and support countless software libraries. GPUs are inherently part of this environment as their primary purpose is to accelerate video rendering. The advent of OpenCL and NVIDIA's CUDA language and tool chains has made GPUs even easier to access for the purposes of general application acceleration. The literature shows a surge of applications accelerated by GPUs since these developments. In contrast, FPGAs have not seen as much developments in accessibility.

FPGAs are flexible enough to emulate custom circuit designs and connect to virtually any device. However, this flexibility also makes it challenging to connect to virtually any device. The protocol standards that make other devices easily interoperable must be included in the FPGA's user design in order for it to interface with external devices. This can be a large obstacle to overcome for application designers. In many cases, implementing the interface logic can match or exceed the effort required for implementing the application logic. As a result, many if not most, FPGA uses involve standalone designs.

This is the motivation that led to the development of RIFFA [5]. RIFFA 2.0 is a reusable interface for FPGA accelerators. Like the original release, RIFFA 2.0 is an open source framework that provides a simple to use interface for software and hardware. It implements the PCIe protocol on both endpoints so that designers can focus on implementing application logic instead of basic connectivity interfaces.

In the sections that follow, we discuss previous work and existing solutions. We also present a detailed description of the RIFFA 2.0 design, example uses, and an analysis of the architecture and performance. This paper's chief contributions are:

- An open source, reusable, integration framework for multi-family FPGAs and workstations.
- An improved hardware and software interface, higher bandwidth, and multi-FPGA support.
- Detailed design for PCIe based DMA bus mastering.

2. RELATED WORK

RIFFA is not the first attempt to integrate FPGAs into traditional software environments. Many research applications exist that solve this problem. However, these solutions are typically highly customized and do not port well to other projects without considerable rework.

Industry offers many solutions for this situation. Impulse Accelerated Technologies, Pico Computing, Convey, Maxeler, and Xillybus all offer products that connect software to FPGAs via a proprietary interface. Their solutions come with software, cores, and some include their own languages, development environments, and tool chains. Many of these solutions exist to drive the purchase of the vendor's goods and services. They are not open source solutions. Nor do they allow users to use their solutions with off-the-shelf components. Moreover, they can be quite expensive. Especially compared to the price of commodity hardware.

There are freely available solutions such as OpenCPI [6] and Microsoft Research's SIRC [7]. OpenCPI is the Open Component Portability Infrastructure project designed to simplify heterogeneous computing. It supports CPUs, DSPs, FPGAs, and other real time embedded devices. As a consequence of this broad support, the setup and configuration of OpenCPI can be challenging. The interface is also overly complex for what is needed for FPGA connectivity. SIRC, a Simple Interface for Reconfigurable Computing, is a Microsoft Research project designed to connect C++ applications to FPGA cores. It is also an open source solution and has been an inspiration for RIFFA. But while SIRC is free, it is only supported on Windows. It also uses a Gigabit Ethernet connection which limits the bandwidth between the host computer and the FPGA. RIFFA uses a PCIe link which offers more scalable performance and is better suited to integrate into workstation, supercomputing, and other high performance computing environments.

Lastly, there are a multitude of FPGA designs that include integrated CPUs. There are also approaches to simplify and allow applications to make better use of FPGA cores such as: Hthreads [8], HybridOS [9] and BORPH [10]. However these solutions utilize custom operating system kernels and often only support CPUs running on the FPGA fabric.

3. DESIGN

RIFFA 2.0 is based on the concept of communication *channels* between software threads on the CPU and user cores on the FPGA. A channel is similar to a network socket in that it must first be opened, can be read and written, and then closed. However, unlike a network socket, reads and writes can happen simultaneously (if using two threads). Additionally, all writes must declare a length so the receiving side knows how much to expect. Each channel is independent and thread safe. RIFFA 2.0 supports up to 12 channels. Up to 12 different user cores can be accessed directly by software threads on the CPU. Designs with more than 12 cores can share channels.

Before a channel can be accessed, the FPGA must be opened. RIFFA 2.0 supports multiple FPGAs per system (up to 5). Each is assigned an identifier on system start up. Once opened, all channels on that FPGA can be accessed without any further initialization. Data is read and written directly from and to the channel interface. On the FPGA side, this manifests as a first word fall through (FWFT) style FIFO interface for each direction. On the software side, function calls support sending and receiving data with byte arrays.

Memory/IO requests and software interrupts are used to communicate between the workstation and FPGA. The FPGA exports a configuration space accessible from an operating system device driver. The device driver accesses this address space when prompted by user application function calls or when it receives an interrupt from the FPGA. This model supports low latency communication in both directions. However, only status and control data is sent using this model. Data transfer is accomplished with large payload PCIe transactions issued by the FPGA. The FPGA acts as a bus master DMA engine for both upstream and downstream transfers. In this way multiple FPGAs can operate simultaneously in the same workstation with minimal system load.

The details of the PCIe protocol, device driver, DMA operation, and all hardware addressing are hidden from both the software and hardware. This means some level of flexibility is lost. For example, users cannot setup custom PCIe base address register (BAR) address spaces and map them directly to a user core. Nor can they implement quality

Function Name & Descramblertion
`int fpga_list(fpga_info_list * list)` Populates the `fpga_info_list` pointer with info on all FPGAs installed in the system.
`fpga_t * fpga_open(int id)` Initializes the FPGA specified by id. Returns a pointer to a `fpga_t` struct or `NULL`.
`void fpga_close(fpga_t * fpga)` Cleans up memory and resources for the specified FPGA.
`int fpga_send(fpga_t * fpga, int chnl, void * data, int len, int offset, int last long timeout)` Sends `len` 4-byte words from `data` to FPGA channel `chnl`. The FPGA channel will be sent `len`, `offset`, and `last`. `timeout` defines how long to wait for the transfer. Returns the number of 4-byte words sent.
`int fpga_recv(fpga_t * fpga, int chnl, void * data, int len, long timeout)` Receives up to `len` 4-byte words from the FPGA channel `chnl` to the `data` buffer. The FPGA will specify an offset for where in `data` to start storing received values. `timeout` defines how long to wait for the transfer. Returns the number of 4-byte words received.
`void fpga_reset(fpga_t * fpga)` Resets the FPGA and all transfers across all channels.

Table 1. RIFFA 2.0 software (C/C++) interface.

of service policies for channels or PCIe transaction types. However, we feel any loss is more than offset by the ease of programming and design.

To facilitate ease of use, RIFFA 2.0 has software bindings for C/C++, Java 1.4+, and Python 2.7+. Both Windows and Linux platforms are supported. RIFFA 2.0's cores support Xilinx Spartan 6, Virtex 6, and 7 Series FPGAs with data bus widths of 32, 64, and 128. All PCIe Gen 1 and Gen 2 configurations up to x8 lanes are supported.

In the next sections we describe the software interface, followed by the hardware interface.

3.1. Software Interface

The interface for the original RIFFA release attempted to impose a call-and-return style execution paradigm for user cores. RIFFA 2.0 does not impose such a model. As a result, the interface on the software side supports just a few functions. The complete RIFFA 2.0 software interface is listed in Table 1 (for the C/C++ languages). We omit the Java and Python intefaces for brevity.

There are four primary functions in the API: open, close, send, and receive. The API supports accessing individual FPGAs and individual channels on each FPGA. There is also a function to list the RIFFA 2.0 capable FPGAs installed on the system. A reset function is provided that programmatically triggers the FPGA channel reset signal. This function can be useful when developing and debugging the software application. If installed with debug flags turned on, the RIFFA 2.0 library and device driver provide useful messages about transfer events. The messages will print to the operating system's kernel log.

There is only one function to send data and one to receive data. This is the basic functionality and is intentionally kept as simple as possible. These function calls are synchronous and block until the transfer has completed. Both take byte arrays as parameters. The byte arrays contain the data to send or serve as the receptacle for receiving data. In these functions, the `offset` parameter is used to specify where in the byte array to start storing data. The `last` parameter is used to group multiple transfers. Multiple transfers may be useful when the FPGA does not have sufficient memory to store all of a computation result. Multiple partial transfers can be issued (with increasing offsets for example) with the `last` parameter set to 0. The software thread won't unblock until `last` is set to 1, which would be set on the final transfer. FPGA cores must be written to honor these uses of the `offset` and `last` parameters to achieve the same behavior in the downstream direction. Lastly, the `timeout` parameter specifies how many milliseconds to wait between communications during a transfer. Setting this value will depend on the timing with which the user core presents data to the channel. Setting a zero timeout value causes the software thread to wait for completion indefinitely.

```
char buf[BUF_SIZE];
int chnl = 0;
long t = 0; // Timeout
fpga_t * fpga = fpga_open(0);
int r = read_data("filename", buf, BUF_SIZE);
printf("Read %d bytes from file", r);
int s = fpga_send(fpga, chnl, buf, BUF_SIZE/4,
    0, 1, t);
printf("Sent %d words to FPGA", s);
r = fpga_recv(fpga, chnl, buf, BUF_SIZE/4, t);
printf("Received %d words from FPGA", r);
// Process results ...
fpga_close(fpga);
```

Fig. 1. RIFFA 2.0 software example in C.

Figure 1 shows an example C application. In this example, the software reads data into a buffer, sends the data as payload to the FPGA, and then waits for a response. The response is stored back into the same buffer and then processed. This example may be trivial, but it represents the canonical use case.

3.2. Hardware Interface

The interface on the hardware side is composed of two sets of signals; one for receiving data and one for sending data. These signals are listed in Table 2. The ports highlighted in red are used for handshaking. Those not highlighted are the FIFO ports which provide first word fall through semantics. The value of `DWIDTH` is: 32, 64, or 128, depending on the PCIe link configuration.

For upstream transactions, `CHNL_TX` must be set high. It must be held high until the channel pulses `CHNL_TX_ACK` high and all the transaction data is consumed. `CHNL_TX_LEN`, `CHNL_TX_OFF`, and `CHNL_TX_LAST` must have valid values until the `CHNL_TX_ACK` is pulsed. The `CHNL_TX_DATA_OFF` value determines where data will start being written in the thread's receiving byte array. This is measured in 4-byte words. As described in the Section 3.1, `CHNL_TX_LAST` must be 1 for the receiver thread to unblock at the end of the transfer. Data values asserted on `CHNL_TX_DATA` are consumed when both `CHNL_TX_DATA_VALID` and `CHNL_TX_DATA_REN` are high.

The handshaking ports are symmetric for both sets of signals. Thus, with downstream transactions, the user core must acknowledge the transaction and consume data from the interface. Timing diagrams for these signals are available on the RIFFA 2.0 website:
`http://cseweb.ucsd.edu/~mdjacobs`.

Figure 2 shows a Verilog example matching the C example code from Figure 1. In this example, the user core receives data from the software thread, counts the number 4-byte words received, and then returns the count.

Signal Name	I/O	Description
CHNL_RX_CLK	O	Clock to read data from the incoming FIFO.
CHNL_RX	I	High signals incoming data transaction. Stays high until all data is in the FIFO.
CHNL_RX_ACK	O	Pulse high to acknowledge the incoming data transaction.
CHNL_RX_LAST	I	High signals this is the last receive transaction in a sequence.
CHNL_RX_LEN[31:0]	I	Length of receive transaction in 4-byte words.
CHNL_RX_OFF[30:0]	I	Offset in 4-byte words of where to start storing received data.
CHNL_RX_DATA[DWIDTH-1:0]	I	FIFO data port.
CHNL_RX_DATA_VALID	I	High if the data on CHNL_RX_DATA is valid.
CHNL_RX_DATA_REN	O	Pulse high to consume value from on CHNL_RX_DATA.
CHNL_TX_CLK	O	Clock to write data to the outgoing FIFO.
CHNL_TX	O	High signals outgoing data transaction. Keep high until all data is consumed.
CHNL_TX_ACK	I	Pulsed high to acknowledge the outgoing data transaction.
CHNL_TX_LAST	O	High signals this is the last send transaction in a sequence.
CHNL_TX_LEN[31:0]	O	Length of send transaction in 4-byte words.
CHNL_TX_OFF[30:0]	O	Offset in 4-byte words of where to start storing sent data in the CPU thread's receive buffer.
CHNL_TX_DATA[DWIDTH-1:0]	O	FIFO data port.
CHNL_TX_DATA_VALID	O	High if the data on CHNL_TX_DATA is valid.
CHNL_TX_DATA_REN	I	High when the value on CHNL_TX_DATA is consumed.

Table 2. RIFFA 2.0 hardware interface.

3.3. Changes from RIFFA 1.0

RIFFA 2.0 is a complete rewrite of the original release. It supports Xilinx Spartan 6, Virtex 6, and 7 Series FPGAs with all PCIe Gen 1 and Gen 2 link configurations up to x8 lanes. The original release is supported on only the Xilinx Virtex 5. RIFFA 1.0 also requires the use of a Xilinx PCIe PLB Bridge core, which has been deprecated. This dependency limits RIFFA 1.0 to x1 lane PCIe Gen 1 configurations. Additionally, due to bus protocol interactions with the PCIe PLB Bridge core, the maximum throughput for upstream and downstream transfers is 181 MB/s and 25 MB/s respectively.

```
parameter INC = DWIDTH/32;
assign CHNL_RX_ACK = (state == 1);
assign CHNL_RX_DATA_REN=(state==2||state==3);
assign CHNL_TX = (state == 4 || state == 5);
assign CHNL_TX_LAST = 1;
assign CHNL_TX_LEN = 1;
assign CHNL_TX_OFF = 0;
assign CHNL_TX_DATA = count;
assign CHNL_TX_DATA_VALID = (state == 5);
wire data_read =
  (CHNL_RX_DATA_VALID & CHNL_RX_DATA_REN);

always @ (posedge CLK)
  case(state)
  0:  state <= (CHNL_RX ? 1:0);
  1:  state <= 2;
  2:  state <= (!CHNL_RX ? 3:2);
  3:  state <= (!CHNL_RX_DATA_VALID ? 4:3);
  4:  state <= (CHNL_TX_ACK ? 5:4);
  5:  state <= (CHNL_TX_DATA_REN ? 0:5);
  endcase

always @ (posedge CLK)
  if (state == 0)
    count <= 0;
  else
    count <= (data_read ? count+INC:count);
```

Fig. 2. RIFFA 2.0 hardware example in Verilog.

RIFFA 1.0 requires users to setup and use Processor Local Bus (PLB) addressing to transfer data. The hardware interface exposes a set of DMA request signals that must be managed by the user core. RIFFA 2.0 exposes no bus addressing or DMA transfer request in the interface. Data is read and written directly from and to FWFT FIFO interfaces on the hardware end. On the software end, data is read and written from and to byte arrays. The software interface has also been significantly simplified.

RIFFA 1.0 supports only a single FPGA per system with C/C++ bindings for Linux. Version 2.0 supports up to 5 FPGAs that can all be addressed simultaneously from different threads. Moreover, version 2.0 has bindings for C/C++, Java 1.4+, and Python 2.7+ on Linux and Windows. Lastly, RIFFA 2.0 is capable of saturating the PCIe link for upstream and downstream transfers. RIFFA 1.0 is not able to achieve more than 73% utilization in the upstream direction or more than 10% in the downstream direction.

4. ARCHITECTURE

On the FPGA, the RIFFA 2.0 architecture is a bus master DMA design connected to a Xilinx Integrated Block for PCI

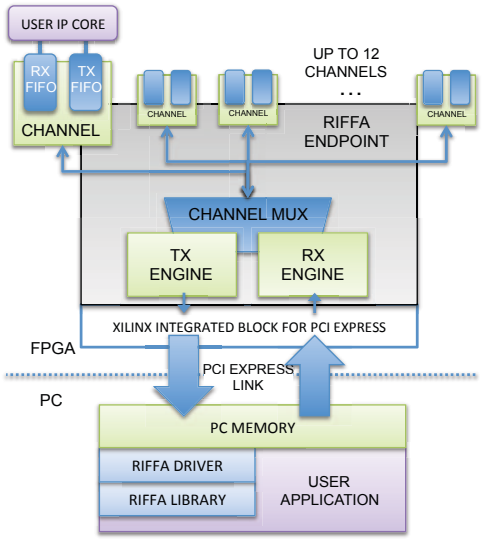

Fig. 3. RIFFA 2.0 architecture.

Express (Xilinx PCIe Endpoint) core. The Xilinx PCIe Endpoint core drives the gigabit transceivers and exposes the PCIe protocol on an AXI bus interface. The AXI bus must be driven using PCIe formatted data packets. The RIFFA 2.0 Endpoint core drives this interface and exposes channels with the RIFFA 2.0 hardware interface for user cores (described in Section 3.2). The RIFFA 2.0 Endpoint is driven by the interface clock; a clock derived from the PCIe reference clock. This clock runs fast enough to saturate the PCIe link. User cores do not need to use this clock for their `CHNL_TX_CLK` or `CHNL_RX_CLK`. Any clock can be used by the user core.

User cores interface with RIFFA 2.0 via a Channel core. The Channel core is written to handle asynchronous clock domains. It has FIFOs for receiving and sending data respectively. To avoid stalling the PCIe link, downstream requests are only made when sufficient space is available in the receive FIFO (RX). Similarly, PCIe upstream transmission is not initiated until sufficient data exists in the sending FIFO (TX).

The RX Engine core is responsible for extracting and demultiplexing received PCIe payload data. The TX Engine core is responsible for formatting payload data into PCIe packets and multiplexing access to the PCIe link. Channel requests are processed in the order they are made. Ties are broken by channel number. This policy prevents any one channel from monopolizing the PCIe link.

The PCIe link configuration determines the width of the data bus. This width can be 32, 64, or 128 bits wide. RIFFA 2.0 supports all three configurations by instantiating different cores for each width. In simpler designs this might just be a parameter to the HDL module. But different bus widths require different logic when extracting and formatting PCIe data. For example, on the 32 bit interface, header packets can be generated one 4-byte word per cycle. Only one 4-byte word can be sent per cycle. However on the 128 bit interface, a single cycle might require formatting three header packets and the first 32 bits of payload. This represents a difference in logic, not just bus width.

On the workstation, the RIFFA 2.0 architecture is a combination of a kernel device driver and a set of language bindings. The device driver is installed into the operating system and is loaded at system startup. It handles registering all detected FPGAs configured with RIFFA 2.0 cores. Once registered, a set of memory buffers are pre-allocated from kernel memory. These buffers will temporarily store data when transferring between the workstation and FPGA. They are allocated so as to be accessible via PCIe. This is sometimes referred to as bounce buffers or a DMA ring. Each buffer is 4 MB in size and the number of buffers allocated depends on how many channels are configured on the FPGAs.

A user library provides language bindings for user applications to be able to call into the driver. The user library exposes the software interface described in Section 3.1. When an application makes a call into the user library, the thread enters the kernel driver and moves data between the pre-allocated buffers. This is accomplished through the `ioctl` function on Linux and with `DeviceIoControl` on Windows.

At runtime, a custom protocol is used between the kernel driver and the Endpoint core. It communicates transfer events such as: when a new transfer is initiated, when a new buffer is needed, or when a buffer is no longer needed. To reduce latency, the protocol uses as few memory/IO PCIe transactions as possible. For example, only three memory/IO writes are needed to to start a downstream transaction.

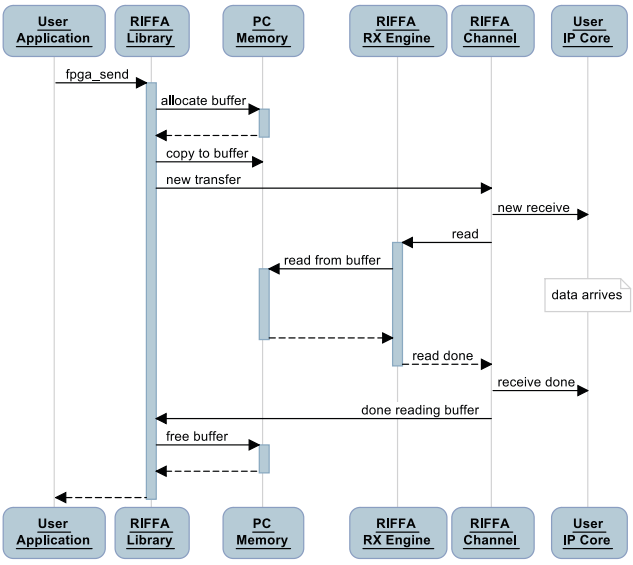

Fig. 4. Downstream transfer sequence diagram.

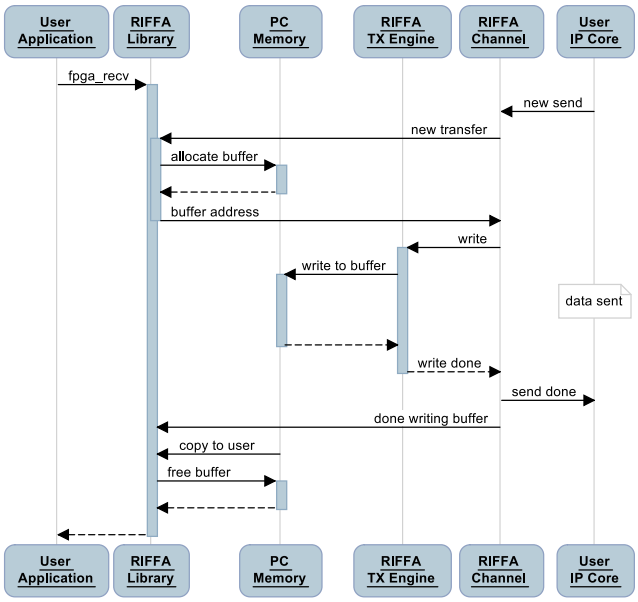

Fig. 5. Upstream transfer sequence diagram.

The Endpoint core sends status information to the workstation using an interrupt. Interrupts spur the driver to read an interrupt vector from the mapped BAR configuration space in the Endpoint. The vector contains events for all channels on the FPGA. Event specifics such as lengths or offsets are read from the Endpoint configuration space in separate memory/IO requests.

The workstation sends status information by writing directly to the Endpoint's configuration space. This can trigger the Endpoint to start transferring data. Data transfer is accomplished using large payload PCIe transactions to maximize throughput. Once a transfer starts, the only communication between the driver and Endpoint is to request new buffers or release used buffers. Both the driver and the Endpoint keep track of how much data is to be transferred so that both are immediately aware of when the transfer ends.

4.1. Data Transfers

A sequence diagram for a downstream transfer is shown in Figure 4. The user application calls the user library function `fpga_send`. The thread enters the kernel driver and acquires a pre-allocated buffer to use as a temporary store for the user data. On the diagram, the user library and device driver are represented by the single node labeled "RIFFA Library". Once a buffer is acquired, data is copied into the buffer so it can be accessed by the Endpoint core. A write to the Endpoint configuration space triggers a new downstream transfer. The write contains the `len`, `offset`, and `last` parameters as well as the address of the kernel buffer containing the data.

Data is read from the buffer into the channel over numerous PCIe transaction layer packets (TLPs). If the data size exceeds a single buffer, the Endpoint core will signal to the driver that it is ready for the next buffer. The driver will acquire another buffer, copy data into the new buffer, and respond with the new buffer address. To improve transfer performance, the Endpoint core will request the next buffer as soon as it recognizes it will need it. This allows the transfer of data in the current buffer to overlap with the filling of the next buffer. This process continues until all the data has been transferred. The release of the last buffer by the Endpoint core signals the end of the transfer to the driver. The driver then frees the last buffer and unblocks the user thread.

A similar sequence takes place for upstream transfers. See Figure 5. The key differences are that the Endpoint core writes data to the kernel buffers and the driver copies the data into the user provided byte array. Additionally, the user core, not the software thread, is the initiator of upstream transfers. This means that data transfer can begin before the user application calls `fpga_recv`. When this happens, the driver will use buffers to store received data until it runs out of buffers or until the user application calls `fpga_recv`. Once the thread enters the driver, data from the kernel buffers can be copied into the user provided byte array.

Lastly, although the sequence diagrams in Figures 4 and 5 use the term "allocate buffer", no runtime allocation takes place. Kernel buffers are pre-allocated at system start up to avoid delays from dynamic memory allocation. The term is meant to describe the allocation of buffers from the pool.

5. PERFORMANCE

We have tested RIFFA 2.0 on three different FPGA development boards with the following configurations.

- AVNet Spartan 6 LX150T
  PCIe x1 Gen 1 link, 32 bit wide data path, 62.5 MHz
- Xilinx ML605 with a Virtex 6 LX240T
  PCIe x8 Gen 1 link, 64 bit wide data path, 250 MHz
- Xilinx VC707 with a Virtex 7 VX485T
  PCIe x8 Gen 2 link, 128 bit wide data path, 250 MHz

RIFFA 2.0 has been installed on Linux kernels 2.6 and 3.1, as well as on Microsoft Windows 7. Our experiments were run on a Linux workstation with quad 3.6 GHz Intel i7 cores using a 12 channel RIFFA 2.0 FPGA design. The user core on each channel was functionally similar to the module in Figure 2. The software was operationally similar to the example listed in Figure 1.

Latency times of key operations are listed in Table 3. Latencies were measured using cycles counted on the FPGA and are the same across all tested boards and configurations. The interrupt latency is the time from the FPGA signaling

of an interrupt until the device driver receives it. The read latency measures the round trip time of a request from the driver to the Endpoint core, and back. The time to resume a user thread after it has been woken by an interrupt is the only value that stands out. At 10.4 μs it is the longest delay and is wholly dependent on the operating system.

Bandwidths for downstream data transfers are shown in Figure 6. The figure shows the bandwidth achieved as the transfer size varies for the three PCIe link configurations. The solid horizontal bars mark the difference between the theoretical maximum for the PCIe link and the maximum achievable bandwidth. PCIe Gen 1 and 2 employ 8 bit/10 bit encoding. This limits the maximum bandwidth achievable to 80% of the theoretical maximum. Our experiments show that we are able to achieve this 80% maximum with sufficiently large transfers on the 32 bit and 64 bit interfaces. The 128 bit interface peaks at 76% utilization.

In Figure 6 you may notice the dip in bandwidths at the 4, 32, and 64 KB transfer sizes for the 32 bit, 64 bit, and 128 bit interfaces respectively. This corresponds to the receive buffer sizes in the Xilinx PCIe Endpoint cores. Looking at the 64 bit interface, we see that bandwidth actually decreases when going from 16 KB transfers to 32 KB transfers. The Xilinx PCIe Endpoint core for the Virtex 6 64 bit interface has a 16 KB receive buffer. Transfers smaller than or equal to 16 KB can actually perform better than transfers with payloads just over 16 KB because there is always buffer space available at the smaller transfer sizes. This artifact becomes negliable when moving larger amounts of data.

The bandwidth figure also shows a slight jump at the 4 MB transfer size for both the 64 bit and 128 bit interfaces (the 32 bit interface is already saturated). This is due to the size of the RIFFA 2.0 kernel buffer being 4 MB. Transfers larger than 4 MB require more than one kernel buffer to hold the data. The time to copy the first 4 MB is seen in the bandwidth curves. However, requests for subsequent 4 MB chunks overlap with the transfer of data from the previous chunk. Thus the copy latency is hidden after the first kernel buffer and bandwidth improves.

While not shown on Figure 6, RIFFA 1.0 was only able to achieve 24 MB/s (10% of max) downstream bandwidth and 181 MB/s (73% of max) upstream bandwidth. This was one of the strongest motivators for RIFFA 2.0.

Resource utilizations for a RIFFA 2.0 Endpoint with a single channel are listed in Table 4. The cost for each additional channel is also listed. Resource values are from

Table 3. RIFFA 2.0 latencies.

Description	Value	
FPGA to host interrupt time	3 μs	$\pm$ 0.06
Host read from FPGA round trip time	1.8 μs	$\pm$ 0.09
Host thread wake after interrupt time	10.4 μs	$\pm$ 1.16

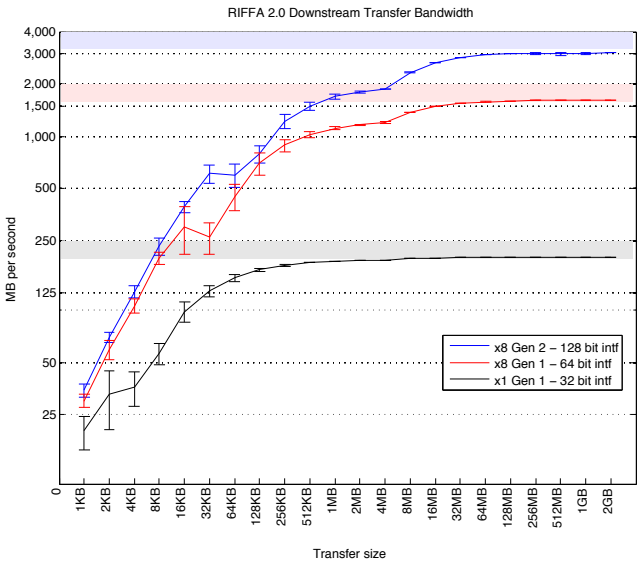

Fig. 6. Downstream transfer bandwidths as a function of transfer size. Upstream bandwidths are nearly identical.

the corresponding FPGA devices and configurations listed above. Wider data bus widths require additional resources for storage and PCIe processing. Single channel designs use less than 1% of the FPGA on all our devices. Even on our most resource limited FPGA, a 12 channel design uses only 18% of the device. The utilizations listed do not include resources used by the Xilinx PCIe Endpoint core. The Xilinx core utilization can vary depending on the configuration values specified during generation. However, the configuration with the highest resource utilization only uses 2237 Slice Registers, 1283 Slice LUTs and 4 BRAMs.

5.1. Factors Affecting Performance

Many factors go into attaining maximum throughput. There is enough confusion on the topic that Xilinx has published a whitepaper [11]. The key components affecting RIFFA 2.0 performance are: transfer size, maximum payload limits, completion credits and receive buffers, user core clock frequency, and data copying.

Table 4. RIFFA 2.0 resource utilization.

RIFFA 2.0 Endpoint with 1 channel	Slice Reg	Slice LUT	Block RAM	DSP 48e
32 bit Endpoint	1657	1814	4	0
addl. 32 bit channel	1092	1458	4	0
64 bit Endpoint	2465	2388	4	0
addl. 64 bit channel	1557	1795	4	0
128 bit Endpoint	3410	3474	8	0
addl. 128 bit channel	1870	2458	8	0

As Figure 6 clearly illustrates, sending data in smaller transfer sizes reduces effective throughput. There is overhead in setting up the transfer. Round trip communication between the Endpoint core and the device driver can take thousands of cycles. During which time, the FPGA can be idle. It is therefore best to send data in as large a transfer size as resources will allow to achieve maximum bandwidth.

When generating the Xilinx PCIe Endpoint core, it is beneficial to configure the Xilinx Coregen Wizard to with the maximum values for payload size, read request size, completion credits, and receive buffers.

The payload size defines the maximum payload for single upstream PCIe transaction. The read request size defines the same for the downstream direction. At system startup, the PCIe link will negotiate a rate that does not exceed these values. The larger the payloads, the higher the bandwidth.

Completion credits and receive buffers are used in the PCIe Endpoint to hold PCIe transaction headers and data. During downstream transfers, completion credits limit the number of in-flight requests that can be made. Receive buffer size limits the amount of data that can be temporarily held. RIFFA 2.0 respects these limits when issuing downstream requests to avoid data corruption and loss. Higher limits provide greater margins for moving data from the workstation to the user core at maximum bandwidth.

Speed of filling and draining the channel FIFOs is also a factor. The user core can be clocked by any source. It need not be the same clock that drives the Endpoint. However, to keep up with the data transfer rate of the Endpoint, it is best for the user core to use the same clock frequency as is used by the Endpoint. Using the same clock is ideal.

Lastly, end-to-end throughput performance can be diminished by excessive data copying. Making a copy of a large buffer of data in software before sending it to the FPGA takes time and can severely impact throughput. The RIFFA 2.0 software APIs accept byte arrays as data transfer receptacles. Depending on the language bindings, this may manifest as a pointer, reference, or object. However, the bindings have been designed carefully to use data types that can be easily cast as memory address pointers and be written or read contiguously.

6. CONCLUSION

We have presented RIFFA 2.0, a reusable integration framework for FPGA accelerators. RIFFA 2.0 provides communication and synchronization for FPGA accelerated applications using simple interfaces for hardware and software. It is an open source framework that easily integrates software running on commodity CPUs with FPGA cores. RIFFA 2.0 extends the original RIFFA project by supporting Xilinx FPGA families: Spartan 6, Virtex 6, and 7 Series. It supports multiple FPGAs in a system, all PCIe link configurations up to x8 for PCIe Gen 1 and 2, and considerably higher bandwidths. It also supports Linux and Windows operating systems with software bindings for C/C++, Java, and Python. We have also provided a detailed analysis of RIFFA 2.0 as a FPGA bus master design and an analysis of its performance. Tests show that data transfers between hardware and software can saturate the PCIe link to achieve the highest bandwidth possible. We hope that users will use RIFFA 2.0 to further the growth of FPGA accelerated applications. RIFFA 2.0 can be downloaded from the RIFFA website at `http://cseweb.ucsd.edu/~mdjacobs`.

7. REFERENCES

[1] R. Kalarot and J. Morris, "Comparison of fpga and gpu implementations of real-time stereo vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, June, pp. 9–15.

[2] R. Spurzem, P. Berczik, G. Marcus, A. Kugel, G. Lienhart, I. Berentzen, R. Männer, R. Klessen, and R. Banerjee, "Accelerating astrophysical particle simulations with programmable hardware (fpga and gpu)," *Computer Science-Research and Development*, vol. 23, no. 3, 2009.

[3] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *SASP*. IEEE, 2008, pp. 101–107.

[4] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with fpgas and gpus," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2010, pp. 115–124.

[5] M. Jacobsen, Y. Freund, and R. Kastner, "Riffa: A reusable integration framework for fpga accelerators," in *Field-Programmable Custom Computing Machines (FCCM), 2012*. IEEE, 2012, pp. 216–219.

[6] J. M. III, "Open component portability infrastructure (opencpi)," 2009.

[7] K. Eguro, "SIRC: An extensible reconfigurable computing communication API," in *FCCM*, R. Sass and R. Tessier, Eds. IEEE Computer Society, 2010, pp. 135–138.

[8] W. Peck, E. K. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews, "Hthreads: A computational model for reconfigurable devices," in *FPL*. IEEE, 2006, pp. 1–4.

[9] J. H. Kelm and S. S. Lumetta, "Hybridos: runtime support for reconfigurable accelerators," in *FPGA*. New York, NY, USA: ACM, 2008, pp. 212–221.

[10] R. Brodersen, A. Tkachenko, and H. Kwok-Hay So, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *CODES+ISSS '06*, 2006.

[11] A. Goldhammer and J. Ayer Jr, "Understanding performance of pci express systems," *White Paper: Xilinx Virtex-4 and Virtex-5 FPGAs*, 2008.