

System-Level FPGA Device Driver with High-Level Synthesis Support

Kizheppatt Vipin, Shanker Shreejith, Dulitha Gunasekera, Suhaib A. Fahmy, Nachiket Kapre

School of Computer Engineering
Nanyang Technological University, Singapore
contact: vipin2@e.ntu.edu.sg

Abstract—We can exploit the standardization of communication abstractions provided by modern high-level synthesis tools like Vivado HLS, Bluespec and SCORE to provide stable system interfaces between the host and PCIe-based FPGA accelerator platforms. At a high level, our FPGA driver attempts to provide CUDA-like driver behavior, and more, to FPGA programmers. On the FPGA fabric, we develop an AXI-compliant, lightweight interface switch coupled to multiple physical interfaces (PCIe, Ethernet, DRAM) to provide programmable, portable routing capability between the host and user logic on the FPGA. On the host, we adapt the RIFFA 1.0 driver to provide enhanced communication APIs along with bitstream configuration capability allowing low-latency, high-throughput communication and safe, reliable programming of user logic on the FPGA. Our driver only consumes 21% BRAMs and 14% logic overhead on a Xilinx ML605 platform or 9% BRAMs and 8% logic overhead on a Xilinx V707 board. We are able to sustain DMA transfer throughput (to DRAM) of 1.47GB/s (74% peak) of the PCIe (x4 Gen2) bandwidth, 120.2MB/s (96%) of the Ethernet (1G) bandwidth and 5.93GB/s (92.5%) of DRAM bandwidth.

I. INTRODUCTION

FPGAs are used in both embedded platforms and specialized, stand-alone, bespoke computing systems (e.g. PCI Pamette [1], Splash [2], BEE2 [3]). The ability to design custom interfaces allows data to be streamed to and from FPGA logic pipelines at very high throughput. We have seen FPGAs make their way into commodity computing platforms as first-class computing devices in tandem with CPUs. Some platforms allow FPGAs to be integrated over a PCIe interface (e.g. Xilinx ML605, VC707, Altera DE4, Maxeler Max3), some over Ethernet (e.g. Maxeler 10G, NetFPGA), and others using a CPU-socket FSB interface (e.g. Convey HC, Nallatech ACP). There have also been recent attempts at creating open-source, standards-inspired interfaces (e.g. RIFFA, OpenCPI, SIRC) which further ease design burden. We investigate the design and engineering of an FPGA driver that (1) is portable across multiple physical interfaces, and (2) provides simple plug-and-play composition with high-level synthesis tools.

In this regard, the stable CUDA driver API is an example of effective driver interface design. It supports a variety of CUDA-capable GPU devices in a high-performance, portable manner. It provides a limited set of interaction primitives that are precise, clear and behave consistently across different GPU devices. In the context of an FPGA driver, we have a harder challenge. We must design both the hardware and software components of the driver. The idea of considering

device drivers as a hardware-software co-design problem has been previously explored [4]. We must worry about diversely-varying FPGA boards, target multiple physical interfaces, provide an interface-agnostic view of communication, have minimal impact on user logic, and consider the impact of device reprogramming on system stability. Our FPGA system-level driver attempts to address some key challenges:

- **Unifying Multiple Physical Interfaces** Existing open-source FPGA drivers typically support a single interface (e.g. RIFFA supports PCIe, SIRC supports Ethernet). We abstract different physical interfaces (PCIe, DRAM, Ethernet) into register-level (AXI-Lite), and handshake-based streaming (AXI-stream) interfaces for integration with user logic generated by high-level synthesis tools. This abstraction simplifies composition with user logic.
- **High-Performance Connectivity** We develop a programmable AXI-compliant switch to allow user logic to send/receive data over multiple physical interfaces. This allow sustained, high-throughput overlapped data transfers to user logic (for some combinations).
- **Driver Safety** Safety and security guarantees have traditionally been a central requirement of device drivers. Unlike some existing solutions, our driver allows the host to reprogram the FPGA without requiring a system reboot or losing PCIe link state. Furthermore, we also provide memory safety on FPGA DRAM in our memory allocation logic and ensure safe concurrent AXI switch transactions.
- **Communication Abstractions** Most existing FPGA drivers support raw PCIe (DMA and PIO) transactions that must be manually adapted to support user logic communication protocols. Some drivers provide bindings to languages like C, C++ or Python which still require low-level management of FPGA logic. Our driver provides direct bindings into HLS test-benches and build environments, enabling a direct, seamless mapping to hardware generated by HLS tools like Vivado HLS, Bluespec, and SCORE. The programmer writes the HLS function and associated test-bench and can switch to FPGA execution when needed.
- **System-Level Environment Monitoring** Apart from communication and reconfiguration support, a true system-level integration is possible if we can monitor, diagnose, and potentially correct environmental operating conditions. We provide support for reading power, voltage, and temperature information from the FPGA.

TABLE I: Survey of existing FPGA system-level drivers.

Driver	Platform	PCIe	DRAM	Ethernet	Config.	Peak B/W	
						Read	Write
SIRC [5]	Windows			✓		118	118
RIFFA 1 [6]	Linux	✓				181	25
RIFFA 2 [7] ¹	Linux/Windows	✓				1600 ¹	1600 ¹
OCPI [8]	Linux	✓	✓		✓	925	925
Xilinx [9]	Linux/Windows	✓				1370	1767
Leap [10]	RHEL Linux	✓				680 ²	339 ²
Maxeler	RHEL Linux	✓	✓		✓	1418	1436
CUDA	Linux/Windows	✓	✓		✓	1425	1515
This paper	Ubuntu Linux	✓	✓	✓	✓	1474	1452

¹64b x8 Gen 1 ²unpublished/unverified data

II. SURVEY

Device drivers enable interactions with peripherals like parallel/serial ports, modems, storage, and network interfaces, in a safe, fair manner. Many interfaces in modern systems require specialized driver software in order to take full advantage of their capabilities and achieve maximum performance, *e.g.* 10-100G network interfaces, SCSI disks, and PCIe-connected accelerators like GPU and FPGA cards. Unfortunately, using FPGAs within host computers has remained challenging due to a plethora of interfaces, diverse user requirements and general apathy from FPGA vendors. Industrial vendors including Maxeler, BEECube, Bluespec, Nallatech, Solarflare, Mercury and others have long provided proprietary interfaces for their own hardware systems. Furthermore, recent engineering effort in academia has attempted to mitigate this system integration issue, as shown in Table I. With FPGA systems, we have the unique opportunity to customize both the software-side (host) and hardware-side (device) interface to enable optimized interaction.

- SIRC [5], provides a software API and requisite hardware interface for communicating between a Windows host PC and an FPGA board. However, it only supports an Ethernet interface thereby limiting achievable bandwidth.
- RIFFA [6] provides a similar interface over PCIe, but PC to FPGA performance is poor due to the use of a PLB to PCIe bridge. RIFFA 2.0 [7] addresses performance issues, but does not support FPGA DRAM access.
- Virtual RC [11] proposes a virtual system layer (software and hardware) to allow designers to target the same design to different boards with minimal memory transfer overhead for large transfers. Only relative overheads are reported.
- The Liquid Metal IP Bridge [12] supports multiple boards. However, it only supports PCIe on some, and achieves a maximum throughput of under 1GB/s, due to the abstraction layer it uses.
- The OpenCPI [8] framework is aimed at supporting interoperability across a range of implementation platforms, including multi-core CPUs, FPGAs and GPUs and software abstractions including OpenCL, and CORBA. Hence, the

framework may be unwieldy and tedious for those seeking to use FPGAs in isolation for a specific design.

The device driver proposed in this paper addresses the weaknesses of previous work, by facilitating interfaces to PCIe, Ethernet, and DRAM, all at high throughput, in addition to supporting multiple access modes to suit a wider range of applications, and bindings to high level synthesis tools to further ease integration and use.

III. INTERFACE DESIGN

In this section, we describe the design of our system-level interface and discuss usage scenarios for the programmer through simple examples.

A. Programmer-Level View

System developers that use GPUs for offloading complex compute tasks will typically identify functions to be implemented in kernels. These kernels run on the GPU while the control program runs on the host. A simple example of this for the NVIDIA CUDA programming environment is shown in Listing 1. In this example, the accelerator kernel is compiled to run on the GPU and called in a wrapper C/C++ program running on the host. The host program is also responsible for data-transfer and invocation of the GPU function with the correct parameters. This CUDA design flow is similar to the strategy used by high-level synthesis users who separate their computation into hardware and software components. The hardware blocks will be implemented on the FPGA while interacting with the software programs or test-benches running on the host CPU. Listing 2 shows how we invoke the driver functions that need to be called within a host program. This mimics CUDA host program behavior (host responsible for data handling and invocation) and offers a simpler path to integrating with existing HLS testing flows.

The entire FPGA design flow using our driver (including hardware generation) is shown in Fig. 1. The bitstream generation process has been suitably adapted to be compatible with our Verilog wrapper interfaces.

B. HLS Communication Abstractions

Modern high-level synthesis tools provide a fresh perspective on compiling high-level descriptions of computation into low-level hardware implementations. This has been made possible by innovations in scheduling and binding within HLS compilers. A key (somewhat overlooked) aspect of this resurgence is the standardization of communication interfaces that move beyond simple busses.

Advanced Microcontroller Bus Architecture - Advanced eXtensible Interface-4 (AMBA AXI-4) is an open standard ARM-developed protocol for high-speed on-chip data communication. Xilinx has adopted the AXI protocol for intellectual property (IP) core interfaces. AXI4-Stream uses a simple handshaking based protocol between a master and slave device in a point-to-point fashion, transferring data on every clock cycle. AXI4-Stream uses simple valid (TVALID), back-pressure (TREADY) handshaking to enable high speed data transfers between AXI cores.

```

1      #include "cuda.h";
2
3      // GPU kernel
4      __global__ void foo(int* a, int* b) {
5          b[threadId.x] = a[threadId.x]+1;
6      }
7
8      // GPU host code runs on CPU
9      int main() {
10         ...
11         // allocate gpu inputs
12         cudaMalloc(device_a, size_a, ...);
13         cudaMalloc(device_b, size_b, ...);
14
15         // copy inputs
16         cudaMemcpy(device_a, host_a, size_a, ...);
17
18         // call GPU kernel
19         foo<<grid, block>>(device_a, device_b);
20
21         // copy outputs
22         cudaMemcpy(host_b, device_b, size_b, ...);
23
24         // free gpu data
25         cudaFree(device_a, ...);
26         cudaFree(device_b, ...);
27         ...
28     }

```

Listing 1: CUDA kernel example.

```

1      #include "fpga.h";
2
3      // FPGA Vivado HLS kernel
4      void foo(int* a, int* b) {
5          *b = *a+1;
6      }
7
8      // FPGA Host code runs on CPU
9      int main() {
10         ...
11         // allocate fpga dram memory
12         device_a = fpga_malloc(size_a);
13         device_b = fpga_malloc(size_b);
14
15         // call FPGA logic
16         load_bitstream("foo.bit",...);
17
18         // copy inputs
19         fpga_transfer_data(HOST, USER1,
20                             device_a, host_a, size_a, ...);
21
22         // copy outputs
23         fpga_transfer_data(USER1, HOST,
24                             device_b, host_b, size_b, ...);
25
26         // free fpga dram memory
27         fpga_free(device_a);
28         fpga_free(device_b);
29         ...
30     }

```

Listing 2: FPGA kernel example.

We now summarize the interface styles generated by Vivado HLS, Bluespec and SCORE:

- **Vivado HLS** Vivado HLS is a high-level synthesis tool based on AutoPilot that converts a restricted subset of C/C++ code to Xilinx FPGA devices and uses the AXI standard for logic interfaces. Our driver directly supports the AXI-Stream subset of the standard.
- **Bluespec** Bluespec is a high-level synthesis tool that supports generation of hardware using a high-level description of computation in a functional programming language. In Bluespec, we describe computation using rules and methods. Method calls are compiled to a hardware interface using an *enable-ready* handshake. With minor modifications to wrapper Verilog (AND masking the enable with the ready), the generated hardware can seamlessly plug into our driver interface.
- **SCORE** Computations described as dataflow operations on streams can be expressed using the SCORE high-level framework [13]. The Verilog generated from SCORE is separated into FIFO-level handshakes on each stream. These handshakes are directly compatible with our driver interface with no modification.

C. Driver Architecture

At a minimum, FPGA drivers must provide the programmer with the ability to load an application bitstream and support data transfers between the FPGA and host. The important engineering decision (and challenge) to consider is how to partition driver functionality between the hardware and software components of our FPGA driver. Certain operations, such as memory management, and performance optimization of Host→FPGA transfers (e.g. double-buffering recipe) are best handled in software, while others, such as low-level DRAM access, handshaking, and buffering, are best done in hardware.

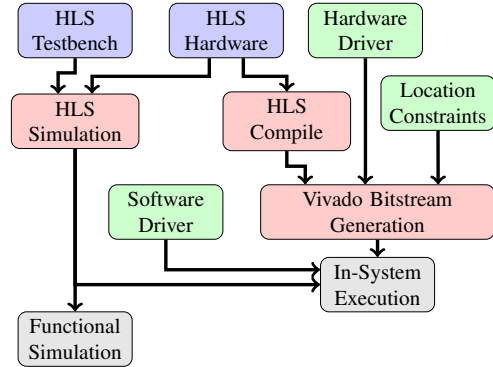


Fig. 1: Programming flow.

We show a high-level block diagram of our FPGA driver in Fig. 2. We now take a deeper look at the organization of our driver.

1) *Software*: In our driver, we build upon the software-side interfaces from RIFFA 1.0 as they provide simple Linux I/O interfaces to support PIO (programmed IO for simple register reads/writes) and DMA (direct memory access for transferring large amounts of data) transfers. RIFFA can only transfer data over PCIe directly to FPGA logic. We substantially expand on this driver in many ways:

- We provide several additional communication pathways to get data to/from the FPGA using the *fpga_transfer_data* API calls. This includes adding paths from PCIe→DRAM as well as an Ethernet→DRAM path. We provide a simple interface selection API.
- The driver supports FPGA DRAM memory management. This provides API calls such as *fpga_malloc* and *fpga_free*

to simplify the task of tracking free unused DRAM space. This allows us to provide memory safety as we no longer permit direct physical address access to user logic streams.

- We program the FPGA using the `load_bitstream` and `fpga_reboot` API call.
- There are separate PIO reads to access system monitor values such as voltage, current, and temperature (ML605 only for now).

2) *Hardware*: Our FPGA driver architecture is co-designed to include hardware and software components that operate in conjunction to deliver efficient and safe operation, as depicted in Fig. 2. While we build upon RIFFA for the software component of our driver, we have developed the hardware component of the driver from scratch for maximum performance and scalability.

- The centerpiece of our hardware driver is the AXI4-Stream switch that provides circuit-switched, bidirectional communication between PCIe \leftrightarrow DRAM, PCIe \leftrightarrow user logic, DRAM \leftrightarrow user logic, and Ethernet \leftrightarrow DRAM. This is unlike the more common packet-switched or time-multiplexed flavors already demonstrated [14] on FPGA overlay architectures. This switch can be programmed at runtime. The switch is rich enough to support concurrent, simultaneous transfers on non-interacting paths. We should note that time-multiplexing is indeed used internally within the arbitration logic (not within the switch) when a specific resource needs to be shared (*e.g.* DRAM \leftrightarrow User Logic).
- Our PCIe interface uses the FPGA PCIe hard macros. The interface is shared by 6 channels: 4 for user logic, 1 each for DRAM and PIO. To enhance DRAM throughput, we use two virtual channels and reordering logic on the FPGA to allow back-to-back transactions. Channel management is an example of partitioning of functionality between the FPGA (reordering logic) and the host (synchronization). We also support non-blocking PCIe transfers with deferred (coalesced) synchronization.
- The Ethernet interface allows near line-rate processing of raw Ethernet frames at Gigabit rates. For raw throughput, OSI layer implementations (TCP/IP, UDP) are mostly irrelevant, so we provide an AXI-compatible interface into the switch.
- User logic is accessible via a programmable number of bidirectional channels (up to 4). Asynchronous FIFOs at the interface allow the user logic to run at a different frequency to the rest of the design. User clocking and reset controls are supported at runtime via higher-level API calls without requiring the user to deal with low-level PLLs and resets.
- Seamless DRAM memory support is missing in a few existing open-source FPGA device drivers (see Section II). Our driver allows access to the onboard FPGA DRAM through a multi-port controller and arbiter. The AXI switch can be configured to saturate 90% of the DRAM memory bandwidth through simultaneous activation of multiple channels.

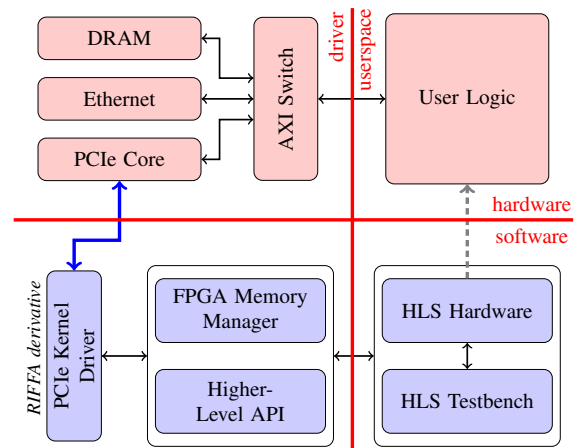


Fig. 2: System-level block diagram.

- Fast runtime reconfiguration of user logic is possible from Platform Flash and BPI Flash interfaces without a host reboot.

D. Usage Scenarios

We highlight four key operating usage scenarios:

- **DMA transfers to off-chip DRAM:** In this mode, we mimic the CUDA `cudaMemcpy()` functionality by staging the offloaded arrays in the FPGA DRAM from the host DRAM via DMA transfers. This mode allows high-throughput transfer of data to a large-capacity off-chip DRAM that can saturate a large percentage of PCIe bandwidth. We operate the transfer in double-buffered fashion to allow us to overlap FPGA computation with subsequent transfers. The interface includes pre-package address generators for different access patterns such as streams (sequential), with the ability to support more in the future (*e.g.* strided). We show a timing sequence diagram for performing a PCIe \rightarrow DRAM data transfer in Fig. 3.
- **DMA Transfers to User Logic:** This is the key functionality provided by competing FPGA open-source drivers like RIFFA and SIRC for streaming data transfers directly to FPGA logic via on-chip BlockRAMs. In this scenario, we have lower latency access to the FPGA fabric but throughput is limited by on-chip FPGA buffering capacity. Presently, we support FIFO-based transfer of data to user logic.
- **Register Read/Write Interface:** In many cases, it is important to quickly synchronize small amounts of control information with the FPGA, *e.g.* start/stop user logic, adjust constant parameters, load runtime configuration parameters for an IP core. For this scenario, we use the PCI programmed IO (PIO) interface to orchestrate these transfers. This same interface is used to support other interface setup operations (*e.g.* PCIe \rightarrow DRAM transfer sizes, AXI switch configuration, system monitor readback).
- **Dynamic Reconfiguration of User Application:** A key advantage of using FPGA fabrics in high-performance computing systems is their ability to be reprogrammed for differ-

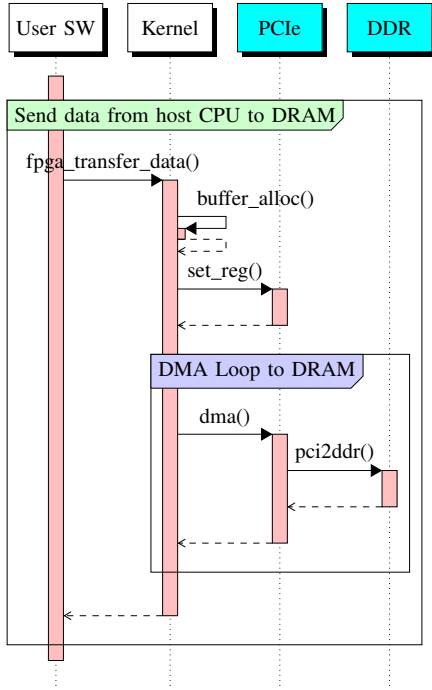


Fig. 3: Sequence diagram for PCIe→DRAM transfer.

ent compute-intensive kernels. For FPGA accelerator cards, a full reprogram has the potential to destabilize the host OS as well as corrupt the DRAM state on the FPGA board. Our driver provides safe caching of PCIe configuration across reconfigurations. This enables us to perform seamless multi-context switching for highly compute-intensive applications.

IV. DRIVER ENGINEERING

We now describe our test infrastructure and the experimental setup used to characterize our driver.

A. Framework Specifications

We use the Xilinx ML605 (XCV6LX240T FPGA) [15] and VC707 (XC7VX485T FPGA) [16] platforms for our experiments. We host these boards in an HP Z420 workstation with an Intel Xeon E5-1650 3.2GHz CPU with 16GB RAM and an Intel X79 PCIe root complex. The PCIe slot used is 8-lane Gen1, 4-lane Gen2, supporting a maximum data rate of 2.5 GB/s with a PCIe link overhead of 0.5 GB/s (8b/10b encoding). We run our experiments in Ubuntu Linux 64-bit 12.04 LTS which supports PCIe hot-plug. We use Xilinx ISE 14.4 to compile our driver. For timing measurement, we use PAPI on the CPU, while on the FPGA, custom timers are used to obtain accurate cycle counts.

B. First-time bootstrapping

To begin using our device driver for the first time, we require a somewhat manual bootstrapping phase at system power-up bitstream load. While this could be achieved in a number of ways, we choose to program the onboard Platform Flash XL (ML605) or BPI flash (VC707) and configure the FPGA from

TABLE II: Framework user APIs.

API Call with Brief Description
System Initialization and Programming
load_bitstream(bitfile, dest_id) Reprogram FPGA through JTAG with <i>bitfile</i> , specifying target device
fpga_reboot(address) Reprogram by loading a bitstream from the external flash using ICAP
fpga_read_sys_param() Read system monitor values such as temperature, voltage, current, power
Data transfer
fpga_transfer_data(src, dest, data, len, addr) Initialize a DMA transfer between <i>src</i> and <i>dest</i> of array <i>data</i> of length <i>len</i> : <i>src</i> : HOST, DRAM, USERPCIE1..4, USERDRAM1..4, ETHERNET <i>dest</i> : HOST, DRAM, USERPCIE1..4, USERDRAM1..4, ETHERNET <i>addr</i> specifies FPGA DRAM address
fpga_wait_interrupt(channel) Synchronization function for data transfers. Channel specifies the specific DMA channel for which synchronization is needed
fpga_reg_wr(addr, data) Write single 32-bit register in global register set or user logic
fpga_reg_rd(addr) Reading single 32-bit register in global register set or user logic
fpga_ddr_pio_wr(addr, data) Indirect write to a single 32-bit DRAM memory location
fpga_ddr_pio_rd(addr) Indirect read from a single 32-bit DRAM memory location
User configuration
user_soft_reset(polarity) Issues a soft reset to the user logic with the specified polarity. Different HLS tools use different reset polarities.
user_set_clk(frequency) Set the clock frequency to the user logic. (250, 200, 150 and 100 MHz)

flash at each power-up. This allows us to perform the bootstrapping once, barring firmware upgrades. Additionally, the kernel driver needs to be compiled and installed. Changes to permissions on the PCIe configuration parameters are required to allow userspace execution of FPGA reconfiguration steps.

C. FPGA Reconfiguration

A lingering misconception among a minority of practitioners is the need to reboot the PC each time the PCIe-endpoint on the FPGA card is reconfigured. This can be avoided by exploiting Linux PCIe hot-plug functionality [17] and restoration of PCIe link parameters after reconfiguration. We save and restore link state on the host during the *load_bitstream()* API call. This functionality is also supported in OpenCPI and LEAP.

D. Driver API and Address Map

In the simple example shown in Listing 2, we highlight the few API calls most likely used by the end-user. We now provide a comprehensive list of API functions calls in Table II. Three types of API support are provided: (1) Data transfer, (2) Reconfiguration, and (3) System monitoring.

Internally, several driver tasks and activity triggers are address-mapped. For example, we program the AXI switch (specify input→output connection) using PIO register writes. The DMA controller configuration is also programmed using address-mapped PIO writes.

E. High-Level Synthesis Workflows

Each of the high-level synthesis tools considered in this paper generates hardware that matches a standard set of IO protocols that are AXI-friendly. To use our driver, we expect the hardware designer to have an existing hardware-generation and functional simulation workflow in place. The driver provides an API to access the physical FPGA as shown earlier in Table II.

- For SCORE and Vivado HLS, the functional simulation will need to be manually modified to target the FPGA backend. For example, in Listing 2, the original functional simulation will simply call the *foo* function with data allocated on the host. The *load_bitstream*, *fpga_transfer_data*, *fpga_malloc* and *fpga_free* calls presently have to be manually written (but should be automatable).
- For Bluespec, test-benches can be synthesizable but there is no straight-forward solution for integrating Bluespec-compiled code into a C/C++ harness (although the other way round is possible). In [18], the authors explore the possibility of automatically performing hardware-software partitioning of Bluespec code. This may be a potential way forward for integration with our driver.

The HLS-generated Verilog must be manually instantiated within a wrapper we supply. For simple interfaces, Verilog assembly can also be automated.

V. CHARACTERIZATION

In this section we characterize the key functional properties of our driver and compare it to some other platforms. We demonstrate the performance achievable over the different interfaces, and discuss technical limitations. Experimental results are shown for the ML605 board but functionality is also tested on the VC707.

A. Driver Hardware Cost

The area usage of the FPGA driver is presented in Table III for the ML605 board and Table IV for the VC707 board. Our driver logic consumes less than 15% of the XC6VLX240T on the ML605 and less than 8% of the XC7VX485T on the VC707 (logic resources), leaving the bulk of the FPGA available for user logic. PCIe endpoint cores, DDR, and Ethernet controllers are generated using Xilinx Coregen. Timing constraints can only be met by locking down the locations of all BRAMs used in the design, especially on the ML605. We should note that certain elements of the driver design are user-configurable (*e.g.* number of PCIe/DRAM channels, Ethernet core inclusion, multiboot support) and can be adjusted to reduce resource usage if desired.

TABLE III: Resource utilization for ML605– 4 user channels (XC6VLX240T device).

Component	Area			Clock (MHz)
	FFs	LUTs	BRAMs	
PCIe Manager	7448	6346	36	250
DRAM Manager	14835	12252	40	200
Ethernet Manager	2780	2527	11	125
Total	25063	21125	87	
(% XC6VLX240T)	8%	14%	21%	

TABLE IV: Resource utilization for VC707 – 4 user channels (XC7VX485T device).

Component	Area			Clock (MHz)
	FFs	LUTs	BRAMs	
PCIe Manager	8101	6130	36	250
DRAM Manager	16401	13799	40	200
Ethernet Manager	4839	3918	11	125
Total	29341	23847	87	
(% XC7VX485T)	5%	8%	9%	

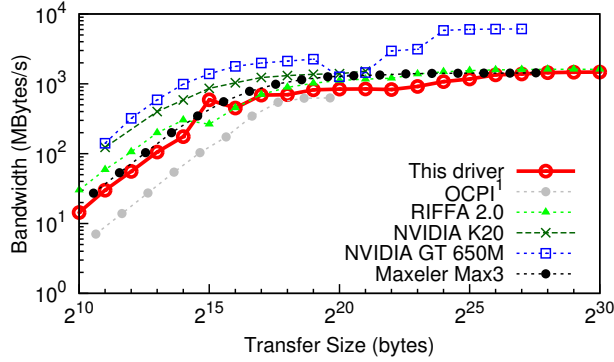
B. PCIe DMA Transfers

Transferring data over the PCIe interface is a cornerstone of the driver’s functionality. This can involve transferring data from the Host to the FPGA DRAM, or to user logic.

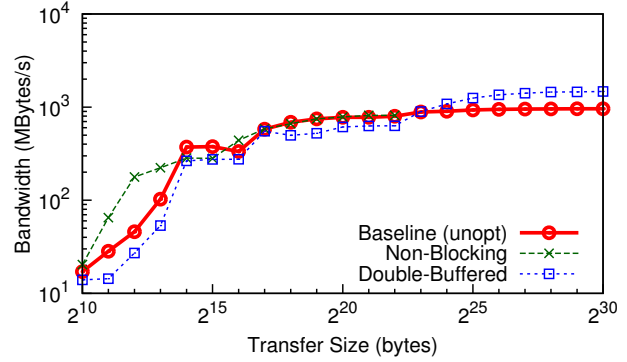
- In Fig. 4(a), we show the achievable throughput for different transfer sizes from Host→FPGA DRAM. For large transfers, throughput peaks at just under 1.5GB/s. Similarly, the reads saturate at 1.45GB/s for large transfers (not shown). Our driver is able to match other drivers’ throughputs (both FPGA and GPU) and only loses to the NVIDIA GT650M due to its superior PCIe Gen3 support.
- Our driver supports direct streaming of data from Host↔FPGA user logic over PCIe, as in the case of RIFFA. Writes peak at over 1.3GB/s, while reads peak at over 1.5GB/s while assuming similar termination at user logic as in [6]. In Fig. 4(b), we see that non-blocking transfers (with deferred synchronization) offer better behavior for small transfers.
- We also support double-buffered writes (via FPGA DRAM) to FPGA user logic which offer a throughput of 1.47GB/s. From Fig. 4(b), we observe that our DRAM-based double-buffered transfer performance matches the direct PCIe transfer performance with the benefit of having access to a larger FPGA DRAM capacity for storage and saving half the copying time.

C. PIO Transfers

Register operations are also supported through the API. Register operations involving on-FPGA registers take 133ns (write) and 1445ns (read). PIO operations to the FPGA DRAM take longer at 264ns (write) and 1785ns (read). If the DRAM controller is busy with refresh/activation on the DRAM banks/rows, we also measure a worst-case latency of



(a) Host→FPGA DRAM PCIe throughput.¹OCPI git commit 9f9ac4.



(b) Other PCIe DMA modes.

Fig. 4: PCIe transfer throughputs.

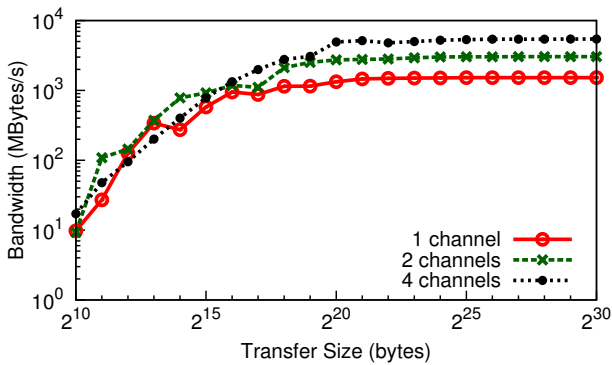


Fig. 5: FPGA user logic→FPGA DRAM throughput

424ns (write) and 1922ns (read). In all cases, reads take longer due the $2\times$ round-trip communication over the PCIe interface.

D. Ethernet Transfers

We conduct throughput experiments on the FPGA Ethernet interface and record a peak bandwidth of 120.24MB/s ($\approx 96\%$ of peak 1G rate) when transferring 64MB of data in either direction. Enhancements to the FPGA Ethernet MAC’s IFG (inter-frame gap) allow higher read throughputs (FPGA→Host) by ensuring a near-constant stream from DRAM to the Ethernet interface.

E. User Logic Transfers

While data can be streamed directly into user logic over PCI, throughputs are limited to ≈ 1.45 GB/s which is comparable performance to Host→FPGA DRAM transfers. Our driver offers improved performance for transfers from user logic to FPGA DRAM (Fig. 5) at a substantially higher data-rate of up to 5.9GB/s when using 4 parallel user channels. The programming flow in such a case is closer to that of CUDA: data is transferred from Host→Accelerator DRAM, the computation is executed on this DRAM data, and DRAM data is read back from Accelerator→Host.

TABLE V: High-Level Synthesis Reports (ML605).

Example	Area			Clk (ns)
	FFs	LUTs	DSP	
<i>Vivado HLS</i>				
32-bit Incrementer	2192	3105	0	4.1
32-bit Squarer	2225	2978	3	4.4
<i>Bluespec</i>				
32-bit Incrementer	2091	1881	0	3.0
32-bit Squarer	2042	1007	3	8.6
<i>SCORE</i>				
32-bit Incrementer	2102	1552	0	3.4
32-bit Squarer	2070	1197	3	8.9

* Area includes 32-deep IO FIFOs in distributed RAM

F. Reconfiguration

When reconfiguring the FPGA over USB/JTAG with uncompressed bitstreams it takes 21 seconds on the ML605, or 17 seconds on the VC707 (the larger bitstream for the V7 is balanced by a faster JTAG clock). Alternatively, multiple bitstreams can be stored in flash memory on the board, and loaded at run time by issuing a reconfiguration command. The time to reconfigure from flash was measured at 120ms on the ML605, and 130ms on the VC707.

G. High-Level Synthesis Interfaces

In Table V, we report the resource usage and clock frequency of the different HLS designs we verified against our driver. We implement the same set of examples in all HLS environments and verify their correctness in simulation as well as a real board. While these examples are small and use simple streaming interfaces, they demonstrate the portability of our underlying driver.

H. Other Drivers

In Fig. 6, we show a Kiviati diagram of resource usage and performance for different FPGA drivers on the ML605. The SIRC, RIFFA 1.0 and RIFFA 2.0 drivers require substantially fewer resources compared to others. While RIFFA 1.0 delivers

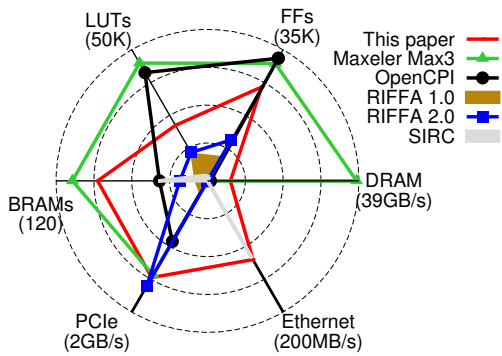


Fig. 6: Comparing FPGA Drivers.

low PCIe throughput, this is rectified in RIFFA 2.0 which exceeds our PCIe throughput by a modest amount. SIRC matches the Gigabit Ethernet line rates achieved by our driver. However, unlike these implementations, our driver provides DRAM support (requiring asymmetric FIFOs with a cost of 24 BlockRAMs) as well as Ethernet connectivity. The more capable OpenCPI driver trades off BRAM capacity for lower PCIe and DRAM throughputs than our driver. Our driver matches the performance of Maxeler for PCIe with a lower resource utilization but is unable to beat the $15\times$ parallel memory banks ($\approx 6\times$ higher bandwidth) in the MAX3 DRAM interface.

VI. DISCUSSION

Considering the aims of this driver, using partial reconfiguration (PR) to separate user logic is an attractive prospect, as it would allow just the user logic bitstream to be generated and loaded as and when needed. However, our attempts with PR raised issues to do with the floorplanning requirements of the design tools [19]. To place the I/O portions of the design (DRAM, PCIe, Ethernet) into a static region, a significant area of the FPGA is reserved by virtue of the spatial arrangement of these pins. As a result, only half the FPGA remains available for user logic, despite the driver not using all the resources in the reserved region. This suggests that hardening full interface drivers (as in the Xilinx Zynq) may be a more promising approach.

VII. CONCLUSIONS AND FUTURE WORK

We have shown how to integrate multiple physical interfaces (PCIe, DRAM, Ethernet, Reconfiguration) on FPGA fabric with hardware logic interfaces generated from high-level synthesis tools (Vivado HLS, Bluespec, SCORE) as well as their corresponding test-bench wrappers running on the host CPU. Our driver consumes less than 15% of logic resource on the ML605 platform while delivering between 74–95% of physical interface throughputs. This FPGA device driver will be released as open source for use by the FPGA community. We intend to expand support for other user logic interfaces, OpenCL bindings, expanded board compatibility, AXI address/data user-level interface (with embedded memory protection), SATA disk interface integration, ICAP readback

for fast BRAM data reloading, and other features in a future release of the driver. The device driver is available for public download from <https://github.com/vipinkmenon/fpgadriver.git> and is compatible with 64-bit Ubuntu 12.04 out-of-the-box.

VIII. ACKNOWLEDGEMENTS

We want to thank Maxeler, Xilinx, Siddhartha, Sagar Masuti, Matt Jacobsen, and Shep Siegel for help in broadening the measurements through equipment access and/or software support.

REFERENCES

- [1] M. Shand, "PCI Pamette user-area interface for firmware v2.0," Compaq Computer Corporation, Tech. Rep., Jun. 1999.
- [2] J. M. Arnold, D. A. Buell, and D. T. Hoang, "The Splash 2 Processor and Applications," *Proc. IEEE International Conference on Computer Design*, pp. 482–485, 1993.
- [3] C. Chang, J. Wawrzynek, and R. Broderson, "BEE2: A High-End Reconfigurable Computing System," *IEEE Design and Test of Computers*, vol. 22, no. 2, pp. 114–125, 2005.
- [4] E. A. Walkup and G. Borriello, "Automatic Synthesis of Device Drivers for Hardware/Software Codesign," in *International Workshop on Hardware-Software Codesign*, 1993.
- [5] K. Eguro, "SIRC: An Extensible Reconfigurable Computing Communication API," in *Proc IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2010, pp. 135–138.
- [6] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A Reusable Integration Framework for FPGA Accelerators," in *Proc. IEEE International Symposium on Field-Programmable Custom Computing Machines*, Apr. 2012, pp. 216–219.
- [7] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *Proc. International Conference on Field-Programmable Logic*, Sep. 2013.
- [8] S. Siegel and J. Kulp, "OpenCPI HDL Infrastructure Specification," Tech. Rep., 2010.
- [9] J. Wiltgen and J. Ayer, "Bus Master Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions," Tech. Rep. XAPP1052, 2011.
- [10] A. Parashar, M. Adler, K. E. Fleming, M. Pellauer, and J. S. Emer, "LEAP: A Virtual Platform Architecture for FPGAs," in *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.
- [11] R. Kirchgessner, G. Stitt, A. George, and H. Lam, "VirtualRC: a virtual FPGA platform for applications and tools portability," in *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2012, pp. 205–208.
- [12] P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "The Liquid Metal IP bridge," in *Proc. Asia and South Pacific Design Automation Conference*, 2013, pp. 313–319.
- [13] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek, "Stream Computations Organized for Reconfigurable Execution," *Journal of Microprocessors and Microsystems*, vol. 30, no. 6, pp. 334–354, Sep. 2006.
- [14] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006, pp. 205–216.
- [15] *UG534: ML605 Hardware User Guide*, Xilinx Inc., 2012.
- [16] *UG885: VC707 Eval. Board for the Virtex-7 FPGA*, Xilinx Inc., 2013.
- [17] A. Piotrowski and D. Makowski, "PCI Express Hot-Plug mechanism in Linux-based ATCA control systems," in *Proc. International Conference on Mixed Design of Integrated Circuits and Systems*, 2010, pp. 148–151.
- [18] M. King, N. Dave, and Arvind, "Automatic generation of hardware/software interfaces," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012, pp. 325–336.
- [19] K. Vipin and S. A. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *Reconfigurable Computing: Architectures, Tools and Applications Proc. International Symposium on Applied Reconfigurable Computing*, 2012, pp. 13–25.