UNIVERSITY OF CALIFORNIA, SAN DIEGO

Smart Frame Grabber: A Hardware Accelerated Computer Vision Framework

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Matthew Daniel Jacobsen

Committee in charge:

Professor Ryan Kastner, Chair
Professor Serge Belongie
Professor Yoav Freund
Professor Rajesh Gupta
Professor Truong Nguyen

2014

The Dissertation of Matthew Daniel Jacobsen is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2014

DEDICATION

This work is dedicated to my family and friends who have supported me throughout my studies. First and foremost, my wife Hayessa, who has put up with endless nights with me at the computer. Also to my close friends Matt and Janet Dowling who have been a constant source of reassurance and opportunity. To my oldest friend Sayf Alalusi and his generous family, who have provided (among many things) an example of how to be. Lastly, to my parents who made this possible.

EPIGRAPH

Great minds discuss ideas;
average minds discuss events;
small minds discuss people.

Eleanor Roosevelt

Nothing is really work
unless you would rather
be doing something else.

J. M. Barrie

Have a good day!
Don’t forget to come home.

Ashton Jacobsen

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

This dissertation is an original intellectual product of the author, Matthew Jacobsen. Research projects not specifically acknowledged as a collaboration are solely the result of the dissertation author.

The work described in Chapter 6 is a collaboration with Pingfan Meng. Mr. Meng was lead researcher on this project. The dissertation author's chief contributions to the project involve the FPGA design and communication framework.

The work described in Chapter 7 is a collaboration with Pingfan Meng and Siddarth Sampangi. The dissertation author was lead researcher on this project. Mr. Meng's chief contribution is the GPU design. Mr. Sampangi's chief contributions are exploration in tracking methods and error quantization.

The work described in Chapter 8 is a collaboration with Siddarth Sampangi. The dissertation author was lead researcher on this project. Mr. Sampangi's contributions are exploration in tracking methods.

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Ryan Kastner for his support as the chair of my committee and my research advisor. His guidance as an advisor and management style have taught me that pursuing knowledge is not only rewarding, but enjoyable as well.

I would also like to acknowledge Professor Yoav Freund for his support as my initial research advisor and committee member. He was instrumental in starting my graduate research career. Without his help, I would not have even begun my research.

Though not my advisor in any official capacity, I would like to thank committee member Professor Truong Nguyen for his help with research projects and opportunities throughout my academic pursuits. His pragmatic view of research has pushed me to achieve results that I could never have imagined.

Chapter 2 contains material as it appears in Field-Programmable Custom Computing Machines (FCCM), 2012. It also contains material as it appears in Field Programmable Logic and Applications (FPL), 2013. The dissertation author was the primary investigator and author of these papers.

Chapter 6 contains material printed in Field-Programmable Technology (FPT), 2012. The chapter also contains material that was omitted from the publication due to space constraints. The dissertation author was not the primary investigator on this paper, but is a the second author.

Chapter 7 contains material printed in Field-Programmable Custom Computing Machines (FCCM), 2014. The chapter also contains material that was omitted from the publication due to space constraints. The dissertation author was the primary investigator and author of this paper.

Chapter 8 contains material printed in Field Programmable Logic and Applications (FPL), 2014. The chapter also contains material that was omitted from the

publication due to space constraints. The dissertation author was the primary investigator and author of this paper.

VITA

1997	Bachelor of Science, University of California, Berkeley
2007	Research Assistant, Department of Computer Science and Engineering University of California, San Diego
2008–2011	Teaching Assistant, Department of Computer Science and Engineering University of California, San Diego
2012	Master of Science, University of California, San Diego
2012–2014	Research Assistant, Department of Computer Science and Engineering University of California, San Diego
2014	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

"Improving FPGA Accelerated Tracking with Multiple Online Trained Classifiers" Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, 2014

"Hardware Accelerated Novel Optical De Novo Assembly for Large-Scale Genomes" Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, 2014

"FPGA Accelerated Online Boosting for Multi-Target Tracking" Field-Programmable Custom Computing Machines (FCCM), 2014 22nd IEEE Annual International Symposium on, 11 May - 13 May 2014

"RIFFA 2.0: A reusable integration framework for FPGA accelerators" Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, 2013

"A hardware accelerated approach for imaging flow cytometry" Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, 2013

"FPGA-GPU-CPU heterogeneous architecture for real-time cardiac physiological optical mapping" Field-Programmable Technology (FPT), 2012 International Conference on. IEEE, 2012

"RIFFA: A reusable integration framework for FPGA accelerators" Field-Programmable Custom Computing Machines (FCCM), 2012 20th IEEE Annual International Sympo-

sium on, 29 April - 1 May 2012

“Detecting, tracking and interacting with people in a public space” In Proceedings of the 2009 international Conference on Multimodal interfaces, November 02 - 04, 2009

FIELDS OF STUDY

Major Field: Computer Science (Computer Vision, Hardware Acceleration)

Studies in Hardware Accelerated Systems
Professor Ryan Kastner

ABSTRACT OF THE DISSERTATION

Smart Frame Grabber: A Hardware Accelerated Computer Vision Framework

by

Matthew Daniel Jacobsen

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor Ryan Kastner, Chair

Real-time computer vision applications have difficult runtime constraints within which to execute. Implementing on a CPU provides a baseline for performance. But using custom parallel hardware such as graphics processing units (GPUs) and field programmable gate arrays (FPGAs) represents a cost effective method to achieve greater performance.

Greater performance can move an algorithm from non-real-time into the realm of real-time. This opens numerous possibilities for interaction that did not exist before. Tasks such as face detection can be used to set focus points in cameras if performed in

real-time. Similarly, body part tracking can be used as input for consumer televisions or video game systems when run in real-time.

Acceleration using heterogeneous hardware is attractive because algorithms exhibit different models of computation at different stages of execution. Each platform can be exploited to execute when most efficient. However, it can be difficult to combine these platforms into a single application. This is due to the lack of reusable components and communication abstractions for these devices.

This work describes a framework to lower the barrier for computer vision application acceleration called the Smart Frame Grabber Framework. This framework is a collection of reusable hardware acceleration components that are commonly used for accelerating computer vision applications using CPUs and FPGAs. It allows applications to be easily partitioned across multiple heterogenous compute devices. At the heart of this framework is a communication and synchronization platform called RIFFA: A Reusable Integration Framework for FPGA Accelerators.

Using the Smart Frame Grabber Framework, researchers can design and build a hardware accelerated computer vision application in considerably less time and with less upfront effort than it would take using existing vendor provided tools alone.

Introduction

Computer vision is the study of acquiring and analyzing images to produce information for the purpose of decision making. Many applications of computer vision have made their way into everyday life. Photo cameras perform face detection to assist automatic focus. Photo organization softwares perform face recognition within their managed collections. Home video game systems and even some televisions detect body parts using cameras and track their locations to provide touch-free interfaces. Many more applications exist in specialized fields such as autonomous vehicles, factory automation, and defense. These applications enrich lives, improve productivity, and even help protect us.

Although this field has seen many advances, there are limitations imposed by the state of the hardware upon which computer vision algorithms would run. These limitations divide the class of applications into two groups: online and offline. The combination of hardware capabilities and algorithm complexity typically make this distinction. Online applications execute quickly enough to produce results with some small amount of latency, usually a few seconds or less. Offline applications have no expectation of producing results quickly and can run for minutes, hours, or longer.

Within online applications, there is a subclass referred to as real-time. This class has the stronger requirement of running several times a second. In the context of human-computer-interaction applications, this requirement is often at least 10 times per second. This is motivated by established human biological thresholds [48, 12]. For

industrial automation this requirement can be several thousand times a second. For example, many factory production lines perform quality assurance on products as they move on conveyers at high speeds.

Regardless of the actual rate, real-time applications run quickly enough to provide decisions immediately. This immediacy makes it feasible to design systems with a feedback loop for interaction. Because the entire loop can be run in real-time, tasks that would be difficult, costly, or impossible to run offline can be achieved. For example, defective products can be dropped from the production line for repair before they are packaged and leave the factory. Similarly, humans can control input to a system sensitive enough to play music [47].

It is the class of real-time computer vision applications that is the most interesting from an interactive perspective. It is also the most challenging to run in real-time. Many algorithms that would be useful in a host of real-time applications simply cannot run fast enough on modern computers. Accelerating these algorithms using custom parallel hardware is an often successful approach for achieving real-time performance. There is however considerable effort required when using this approach. Many of the facilities provided by computers and software are not available when using custom hardware. As a result, most work in this area requires considerable investment in basic infrastructure to even begin. Moreover, the completed designs are frequently not reusable across applications. There is no common framework for hardware accelerating computer vision applications.

This dissertation describes the results of my work to remedy this situation. It describes the Smart Frame Grabber Framework, a computer vision acceleration framework that allows applications to easily leverage CPUs, field programmable gate arrays (FPGAs), and GPUs with minimal rework. This merges research from: vision, hardware acceleration, and high performance computing. It reduces the barriers to entry for accel-

erating computer vision applications with custom hardware. Lastly, it allows researchers and practitioners to focus on application logic instead of common functionality.

The rest of this document is organized as follows. Chapter 1 describes the Smart Frame Grabber Framework in detail and provides motivation. Chapter 2 describes our work in developing the RIFFA communications platform, a core component in the Smart Frame Grabber Framework. Chapters 3, 4 and 5 describe early skin color detection, tracking, and object detection applications built using the Smart Frame Grabber Framework. Chapter 6 describes an optical mapping application cardiac physiology that is partitioned across a CPU, GPU, and FPGA. It also leverages RIFFA and components in the Smart Frame Grabber Framework. Chapter 7 describes and compares two designs for hardware accelerated online boosting for tracking. One is a CPU-FPGA design, the other is a CPU-GPU design. Chapter 8 describes an algorithmic improvement to online boosting for tracking made only feasible because of the performance increase resulting from acceleration. Future directions based on our work with the Smart Frame Grabber Framework are provided in Chapter 9. Detailed descriptions of the Smart Frame Grabber components (including RIFFA) are provided in the appendices.

Chapter 1

Smart Frame Grabber

The Smart Frame Grabber Framework is a collection of reusable hardware acceleration components that are commonly used for accelerating computer vision applications using CPUs and FPGAs. Using this framework, researchers and practitioners can design and build a hardware accelerated computer vision application that easily leverages multiple heterogenous devices. In our work, we have used the framework to integrate CPUs, FPGAs, and GPUs all within the same application in considerably less time and with less upfront effort than it would take using existing vendor provided tools alone.

Algorithms exhibit different patterns of computation during execution. Some parts are highly sequential, others are completely independent and amenable to parallelization. Using multiple heterogenous devices allows each part to run on the device most efficient at that type of computation.

Consider three of the most common computation platforms used. CPUs excel at Von Neumann model sequential processing and rich programming abstractions and libraries make it the easiest platform to program. GPUs can compute two to three orders of magnitude faster than CPUs if there is sufficient independent data to process. However, GPUs are more difficult to program due to their single instruction multiple data (SIMD) execution model. Finally, FPGAs can be designed to emulate any circuit. They often perform very well when configured as stream processors because data is

Table 1.1. Platform characterization for CPUs, GPUs, and FPGAs.

	CPU	**FPGA**	**GPU**	**Smart Frame Grabber**
Processing units:	˜4	˜100	˜1000	˜1000
Cycle frequency:	GHz	MHz (low)	MHz (high)	GHz
Programming:	High level	Low level	Medium level	Low - high level
Reusable libs:	Countless	Few	Few	Countless
Dev. cycle:	Short	Long	Short	Short
Parallelism:	Opportunistic	Fine grain, dedicated	Massive	Massive, dedicated
Pipelining:	Sporadic	Fully	Mostly	Fully
Scheduler:	Dynamic	Custom	Dynamic	Custom
Memory:	Unlimited	Small	Large	Unlimited
Memory latency:	Moderate	Low	High	Low
Data access:	Fetch	Stream or fetch	Fetch	Stream or fetch
Data path:	Fixed	Custom	Fixed	Custom
Computation:	MPMD	MPMD	SPMD	MPMD
Resource growth:	Slow	Fast	Moderate	Fast
Reconfiguration:	Compile time	Run time	Compile time	Run time

manually scheduled and operations are typically executed by dedicated hardware. FPGA's drawback is also difficulty in programming, as compared to CPUs. Table 1.1 lists salient characteristics of these platforms.

Each platform has its strengths and weaknesses. For application acceleration purposes, it is ideal to leverage the strengths of all three platforms. This would allow one application to take advantage of the best that each platform has to offer. This is represented in the Smart Frame Grabber column of Table 1.1.

The difficulty in achieving this ideal is two fold. First, one must efficiently integrate all the devices together in one application. For this task, efficiency is paramount. Loose coupling with low bandwidth communication defeats the purpose of leveraging each device for high performance computing. The second difficulty deals with programming the devices themselves. FPGAs, for example, are difficult to program because of their low level abstractions. However when designed well, they can be extremely powerful devices. The Smart Frame Grabber Framework addresses both of these problems.

First consider application integration. Tight CPU-GPU integration has been

solved by the GPU software development kits provided by manufacturers, such as NVIDIA's CUDA SDK. However CPU-FPGA integration is not as easily accessible. This was a problem that needed to be solved, and solved in a way that was reusable and flexible enough to support multiple FPGA and mixed environment integration (e.g. CPU-GPU-FPGA). The Smart Frame Grabber Framework solves this problem. It does so while also allowing each device to be programmed independently. Any tool or language can be used to program these devices. The framework does not impose a single language or runtime environment as other solutions have. It integrates well with existing CPU based runtime libraries so that it can take advantage of other frameworks like CUDA. This helps preserve the performance characteristics of each platform and increases general applicability.

The Smart Frame Grabber Framework also helps ease the difficulty of programming these devices. In particular, FPGAs. It includes reusable components that are commonly used to perform operations in computer vision applications. These operations include: image scaling, sliding window pipelines, convolution templates, and a host of flow control modules. These components target FPGAs because they are the most challenging to program. Having a set of reusable computer vision oriented components greatly reduces the time to design and build image processing logic in FPGAs.

1.1 Motivation

The idea for the Smart Frame Grabber originated after designing and building several hardware accelerated vision applications. The first application was a skin color detection application. It uses a boost trained decision tree to classify image pixels as human skin. Software running on a CPU controls processing on an attached FPGA. The FPGA captures video frames from a camera, classifies the pixels in a streaming fashion, and renders the image with annotations to a monitor. The software provides parameters to

the FPGA, such as the classifier decision tree, and receives skin pixel counts for specified regions. This application is used in a game that identifies when uncovered human body parts (mostly hands and face) have entered specified regions.

The second application was also partitioned between a CPU and FPGA. It tracks multiple targets in video. Video is captured on the FPGA, processed, and the target locations are returned to the CPU. The software on the CPU updates the location and target template using a particle filter search algorithm. The cost function for each location is the sum of absolute difference between a template and the current window in the frame. This application was used to track up to 6 independent targets in multiple scales at 60 frames per second.

The next project was a hardware accelerated face detection application, again partitioned between a CPU and FPGA. By this point, it was clear that the model of computation was very similar across all these applications. For each frame, each application needed to capture video on the FPGA, transfer parameter data from the CPU to the FPGA, extract features or process image data on the FPGA, and return results to the CPU. Each application had a different set of common components that had been rebuilt several times. Clearly starting each application from scratch was inefficient. Thus the idea of the Smart Frame Grabber Framework was born.

1.2 Communication Component

The Smart Frame Grabber is built around a communications framework called RIFFA: A reusable integration framework for FPGA accelerators. RIFFA provides high bandwidth, low latency communication and synchronization between FPGA devices and computers equipped with a PCI Express (PCIe) connection. It provides this via simple software APIs and a FIFO hardware interface. The chief benefit of RIFFA is that it hides the complexity of transferring data over PCIe behind simple interfaces while providing

efficient use of the PCIe link.

One of the most common bottlenecks in high performance computing is moving data between computing devices. RIFFA is designed to be as efficient as possible while still providing a high level abstraction for users. Latencies are on the order of 1 μs and transfer bandwidths between the CPU and FPGA can achieve 3.6 GB/s, which is 90% of the theoretical maximum bandwidth for the PCIe link. RIFFA isn't a substitute for on chip or even off chip local RAM as PCIe latencies are much higher than when accessing RAM. It is an interconnect between a host CPU and multiple FPGA devices so that multiple devices can be used in a single application. The current version of RIFFA supports FPGA devices from Xilinx and Altera.

RIFFA is also designed to be easily reusable. No refactoring, adjustments, or adapting is necessary between projects. Even many hardware changes are automatically accommodated. RIFFA is only dependent on the FPGA device family. It works like a reusable black box component that performs scatter gather DMA transfers. Most importantly, it does this without requiring the user learn and program a complex DMA interface.

RIFFA is an open source project. It is freely available to researchers and for non-commercial use. The open source nature of the project allows researchers to make enhancements to the architecture as well as learn from a high performing DMA design. As of this writing, RIFFA has been cited by 9 published works and has been used in projects at 14 different universities in Europe, the United States, East Asia, and India.

Documentation for RIFFA is provided in Appendices A and B. This documentation is included to provide users with a single source for RIFFA related documentation.

1.3 Reusable Components

There is a general lack of libraries for FPGAs when it comes to vision processing. Available solutions from vendors are often tied to expensive licenses or specific hardware. Open source components are often buggy and sporadic in terms of coverage. Solutions that leverage higher level constructs like OpenCL help alleviate this situation, but at the expense of predefined interfaces and a rigid runtime environment.

The lack of reusable FPGA libraries is not just the result of a lack of attention. Unlike CPU or GPU programming, there is no standard memory hierarchy. Therefore, data access must be scheduled manually. For efficiency reasons, data is typically streamed though a data path instead of stored in a large global memory and fetched out of order. Formulating sequential operations with random access patterns to work in this streaming manner can be complex and require considerable effort.

As an example, consider a common operation in vision processing, image resizing. On the CPU and even on the GPU, one can simply define the function necessary for bi-linear interpolation by addressing data in an array. To achieve the same on the FPGA, data must be temporarily stored in local on chip memory buffers. The interpolation function executes against local memory in a manually scheduled pipeline. At the same time, new incoming data must be captured while interpolated data is outputted. Doing these steps, while at the same time managing data flow control, is a daunting task.

Another reason FPGA libraries are not pervasive is because of the inherent difficulty in composition. Unlike software on a CPU, combining multiple FPGA modules requires care in order to meet execution timing requirements. This extra dimension forces modules to define not only what must be computed, but also how. Commonly, modules are defined to meet timing requirements when used in a specific context. But when combined with other modules in a new context, may violate these requirements. This is

due to the physical limitations of having to route signals across an FPGA device.

Lastly, reusable FPGA libraries are difficult to come by because there is no established common interface. Components are so diverse that trying to impose a common interface is untenable. High level synthesis applications, such as Xilinx Vivado HLS solve this problem by using FIFO buffers between discrete code blocks. This decouples different parts of a pipeline and produces a common interface between them. It is however inefficient and resource expensive to take this approach.

The reusable components in the Smart Frame Grabber Framework are provided as a toolbox of modules that provide several operations related to computer vision processing on FPGAs. They were developed for a specific project, but added to the framework because of their applicability to other computer vision applications. They have been written to support flow control and be used in pipelined processing. As a result, these components can be used in conjunction with modules from other designs and help reduce the amount of effort needed when building hardware accelerated computer vision applications.

The components in the Smart Frame Grabber Framework include:

- frame capture
- image scaling
- integral image transform
- sliding window processing (convolution)
- mean and variance calculation
- normalized cross correlation calculation
- Haar feature calculation

- skin color detector and generator
- color space conversion
- data buffering and flow control related modules.

The details of each component is provided in Appendix C.

1.4 Contributions

The primary contributions of this work are enumerated below.

1.4.1 Framework Contributions

These contributions relate to the Smart Frame Grabber framework, it's components and design.

RIFFA Scatter Gather DMA Design

The RIFFA architecture is a high performing scatter gather DMA design over PCIe. It is capable of achieving 90% theoretical link bandwidth. It supports multiple FPGA device families and all PCIe Gen 2 configurations. It exposes a simple API in software with bindings for C/C++, Java, and Python. On the hardware side data is read and written to a FIFO interface. RIFFA supports multiple FPGAs per host CPU and can easily integrate into existing hardware and software applications.

This work has been published and is described in Chapter 2.

RIFFA Open Source Software

This includes the HDL source code, Windows and Linux drivers, sample applications, installation packages, test cases, C/C++, Java, Python, and Matlab language bindings, and various configurations required for producing a packaged installable framework.

This work is available on the RIFFA website and is described in Chapter 2.

RIFFA Documentation

This includes documentation provided in this dissertation as well as the RIFFA website. This is composed primarily of software API documentation, hardware interface and timing diagrams, device setup and configuration guides, and best practice guidelines.

This work is available on the RIFFA website and is documented in Appendices A and B.

Smart Frame Grabber Reusable Component Software

This includes the HDL source, test cases, and documentation (provided in this dissertation and with the components).

This work documented is in Appendix C.

1.4.2 Application Contributions

These contributions are of accelerated computer vision applications that have been made possible by using the Smart Frame Grabber framework.

FPGA Accelerated Skin Color Detection

This work accelerates evaluation of a boosted alternating decision tree classifier. The classifier is trained to classify pixels as human skin colored. The FPGA captures frames, evaluates the classifier, and counts skin colored pixels in runtime updated regions of the frame. These values are provided to the CPU each frame at a rate of 60 frames per second.

This is early work that helped contribute to the Smart Frame Grabber framework. It is described in Chapter 3.

FPGA Coprocessor for Particle Filter Tracking

This work partitions a particle filter like tracking algorithm over a CPU and FPGA. The FPGA captures video and evaluates a template based classifier for each target location within the frame. Each location samples windows from a particle distribution and returns the classification scores to the CPU. Up to 6 independent targets can be tracked concurrently at 60 frames per second using only 25 - 40% of the CPU. This represents a 30× speed up over a CPU only implementation.

This is early work that helped contribute to the Smart Frame Grabber framework. It is described in Chapter 4.

FPGA Accelerated Face Detection

This is an application of accelerated face detection using the Viola and Jones algorithm [58]. The application runs nearly all on the CPU, but image acquisition, integral image conversion, and partial classifier evaluation is performed on the FPGA. The partial evaluation results are used as a filter for the software evaluation cascade to reduce workload. The FPGA acceleration is also integrated into the highly optimized OpenCV software library. The acceleration provides a 3.16× speed up over software only.

This is early work that helped contribute to the Smart Frame Grabber framework. It is described in Chapter 5.

FPGA-GPU-CPU Heterogenous Architecture for Real-time Cardiac Physiological Optical Mapping

This work accelerates an optical mapping application for cardiac physiology to run in real-time. Real-time in this application is 1000 frames per second. The application performs image normalization of photovoltaic optical flow video. The design leverages stream processing on a FPGA, high bandwidth GPU processing, and CPU bookkeeping

within the same application. It is able the to process video in real-time at 1024 fps with an end to end latency of 1.86 seconds. This represents a 273× speed up over a multi-core CPU OpenMP implementation.

This work has been published and is described in Chapter 6.

FPGA and GPU Accelerated Implementations of Online Boosting for Tracking

This work accelerates an adaptive online boosted tracking algorithm [7] using a GPU and a FPGA in two separate designs. Speed ups over the highly optimized software-only C++ implementation are 2.7× for the GPU design and 68× for the FPGA design. The FPGA design is capable of tracking 57 independent targets at 30 FPS. This is the first FPGA and GPU accelerated implementations of online boosting for tracking. This work also contributes an analysis of two hardware acceleration platforms and identifies fundamental differences that contribute to performance disparities.

This work has been published and is described in Chapter 7.

Hardware Enabled Multiple Classifier Algorithm for Online Trained Tracking

This work describes a FPGA-CPU accelerated design for tracking objects through appearance changes, using multiple online boosted classifiers. The work presents an algorithm for learning a pool of pose-specific and tracking classifiers at runtime. It also employs a novel method for comparing multiple classifier scores using a kurtosis of the score distributions. Compared to a multi-threaded software-only CPU based implementation, the accelerated implementation boasts a 30× speed up over a highly optimized C++ implementation. This work performs at state of the art levels and shows an improvement in accuracy over existing tracking algorithms.

This work is in submission and is described in Chapter 8.

Part I

Smart Frame Grabber Framework

Chapter 2

RIFFA: A Reusable Integration Framework for FPGA Accelerators

2.1 Introduction

FPGAs and GPUs have become popular parallel computing platforms for application acceleration. Both have been successfully applied to accelerate numerous vision [42], physics [55], and other compute intensive applications [15]. They are even used in heterogenous computing environments for high performance computing [57]. 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.

Our goal is to lower the barriers for application acceleration using FPGAs. To that end, we introduce RIFFA: A reusable integration framework for FPGA accelerators. RIFFA is an integration framework for connecting IP cores on an FPGA with software running on a computer. The framework requires a PCIe bus enabled workstation and a FPGA with a PCIe peripheral. RIFFA provides communication and synchronization capabilities with a standard interface for both software and hardware. It is comprised of Verilog IP cores, software libraries, and a device drivers. RIFFA is also open source so that researchers 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 1.0 and RIFFA 2.0 designs, 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 simplified hardware and software interface, offering high bandwidth, low

latency, and multi-FPGA support.

- A detailed design for PCIe based DMA bus mastering.

2.2 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 [35] and Microsoft Research's SIRC [25]. 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 [51], HybridOS [43] and BORPH [10]. However these solutions utilize custom operating system kernels and often only support CPUs running on the FPGA fabric.

2.3 RIFFA 1.0

The initial version of RIFFA [38] is based on a set of components provided by Xilinx. It relies on a PCIe Endpoint, a PCIe Bridge, and a DMA core available in Xilinx's Embedded Development Kit. It targets Virtex 5 and Virtex 6 devices and supports a single FPGA per host PC.

2.3.1 Architecture

RIFFA 1.0 is a C software library and Linux device driver, on the workstation side, and set of IP cores on the FPGA side. The two are connected via a PCIe bus connection. A diagram of the RIFFA 1.0 architecture is displayed in Figure 2.1.

In designing RIFFA we sought to expose interfaces general enough for most applications, to support high communication throughput with low latency, and to be compatible with off the shelf workstations and FPGAs. For these reasons, we built our framework to use a PCIe bus. PCIe buses are common in most workstations and increasingly so in embedded systems. They offer high bandwidth connections with extremely low latency. Many FPGA boards come equipped with PCIe connections and chip makers are combining FPGAs with CPUs, connected by PCIe, in the same package

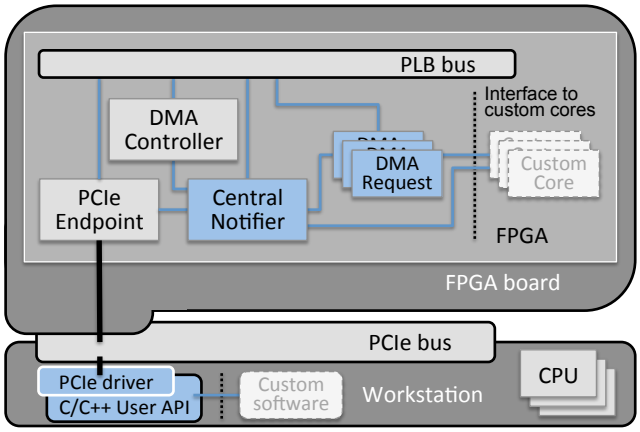


Figure 2.1. Architecture of RIFFA 1.0 framework. Application acceleration cores interface with the DMA Request and Central Notifier cores.

or on the same die[1]. We chose Linux because it is an open source platform with wide adoption and well suited for high performance application execution. Our initial version has targeted only Xilinx FPGAs. Future versions may include support for FPGAs from other vendors.

Software Interface

On the software end, we developed a PCIe Linux device driver and a set of software libraries. The device driver probes for the FPGA at boot time and assigns addresses within the workstation's PCIe address space for the PCIe Endpoint on the FPGA. During this process kernel address space is reserved for communicating with the FPGA. Once address space is assigned, the driver can access the PCIe Endpoint. In order to enable access outside of the kernel, the driver creates a virtual device file in the *dev* filesystem. This virtual device file can then be opened, read from, written to, or memory mapped by any application in user space. In this way, we expose the FPGA to user space. Accessing this virtual device file executes PCIe read or write transactions over the PCIe bus. On the FPGA, the PCIe Endpoint services these requests by translating them to

[1] Intel ECx5C Series and Xilinx Zync platforms.

Processor Local Bus (PLB) requests via address translation. This gives applications on the workstation the ability to access individual IP cores on the FPGA using file operations or memory assignments.

To provide signaling of events, the driver establishes an interrupt channel between the workstation and PCIe Endpoint on the FPGA. Received interrupt vectors identify which IP core has signaled the interrupt. Our driver acknowledges interrupts and exposes individual interrupts by creating a set of numbered virtual files in the *proc* filesystem. When an application attempts to read or poll[2] any of these files, the driver returns the number of interrupts received from the corresponding IP core. If no interrupts have yet been received, the driver sleeps the calling thread and wakes it up when the appropriate interrupt is received. This design exposes many logically distinct interrupt channels using the single PCIe device interrupt in a thread efficient manner. RIFFA currently supports up to 16 interrupt channels. One drawback to this design is that interrupts must be acknowledged by the driver before another interrupt vector can be sent by the FPGA. We mitigate this problem by AND'ing pending interrupt requests on the FPGA so that a single PCIe interrupt received on the workstation can trigger multiple logical interrupt channels.

Even with a fast processor, we found that writing 32 bits of data at a time via PCIe transactions is inefficient for sending more than a few words of data. Additionally, there is no standard software facility for sending interrupts to PCIe devices from the workstation. We therefore added DMA transfer support and workstation-to-FPGA interrupts (so called "doorbells") using PCIe write transactions. Software initiated writes to a controller IP core signal the request for a DMA transfer and/or an interrupt to a specific IP core. This makes it convenient for applications to have input data DMA transferred to the appropriate IP core then have the core interrupt signaled. These IP core doorbells, manifest as line

[2]The *poll* operation is used for asynchronous I/O.

```
void main() {
    fpga_dev_t fpgaDev;
    int intFd, offset, value;

    fpgaMapMemory(&fpgaDev);
    intFd = fpgaInterruptOpen(IP_CORE_NUM);
    ...
    fpgaWriteWord(fpgaDev.dmaMem, offset, value);
    ...
    fpgaFireInterrupt(intFd);
    fpgaInterruptWait(intFd);
    ...
    value = fpgaReadWord(fpgaDev.dmaMem, offset);
    ...
    fpgaInterruptClose(intFd);
    fpgaUnmapMemory(&fpgaDev);
}
```

Figure 2.2. Example usage of RIFFA 1.0 from a user application.

pulses to the receiving IP cores.

Because utilizing this communication and event signaling framework would require understanding of its implementation, we created a high level API for user applications. This library is written in C. Many of the core functions and their descriptions are listed in Table 2.1. A use case example is shown as a call diagram in Figure 2.3. This diagram illustrates the relationship between software function calls on the workstation and hardware signaling on the FPGA. A typical user application would initialize the FPGA connection using $fpgaMapMemory$. Data can then be read and written or DMA transferred using the provided functions. For each interrupt it wishes to wait for, the application must call $fpgaInterruptOpen$. Calls to $fpgaInterruptWait$ block the calling thread until an interrupt is received. Because there are 16 separate channels, multiple threads can wait on different channels without interference. No user level synchronization primitives are needed. When no longer needed, the FPGA connection can be closed using $fpgaUnmapMemory$ and interrupt notifications can be terminated using

Table 2.1. RIFFA 1.0 core software functions.

Function	Description
fpgaMapMemory	Opens the FPGA virtual device file and maps the PCIe address ranges into memory.
fpgaUnmapMemory	Unmaps all memory mapped address and closes the FPGA virtual device file.
fpgaInterruptOpen	Gets the IP core interrupt file descriptor.
fpgaInterruptClose	Releases the IP core interrupt file descriptor.
fpgaInterruptWait	Causes the calling thread to wait until the corresponding IP core fires an interrupt.
fpgaFireInterrupt	Fires an interrupt to the specified IP core.
fpgaRequestDma	Triggers a DMA transfer between the FPGA and workstation memory.
fpgaReadWord	Reads a 32 bit word from FPGA memory.
fpgaWriteWord	Writes a 32 bit word to FPGA memory.

$fpgaInterruptClose$. All the communication details and kernel structures are hidden from the user application. This example is listed in Figure 2.2. Additional APIs are being developed to provide support for common application tasks.

One of the drawbacks of this design is that the kernel driver must be configured with an address space large enough to support the maximum amount of contiguous response data expected from any IP core. We currently set this to 8 MB for our example applications, but plan to address this constraint in future versions.

Hardware Interface

The key hardware components in RIFFA are the PCIe Endpoint, DMA Controller, Central Notifier, and DMA Request cores as pictured in Figure 2.1.

The PCIe Endpoint drives the PCIe slot on the FPGA board. It also functions as a PLB to PCIe bridge so that address space can be mapped between the two buses. This IP core is provided free of charge by Xilinx. It is configured with two 4 MB IPIF-to-PCIe base address register (BAR) mappings and one 8 KB PCIe-to-IPIF BAR mapping. IP interface (IPIF) refers to the PLB side of the PCIe bridge. The IPIF-to-PCIe BARs

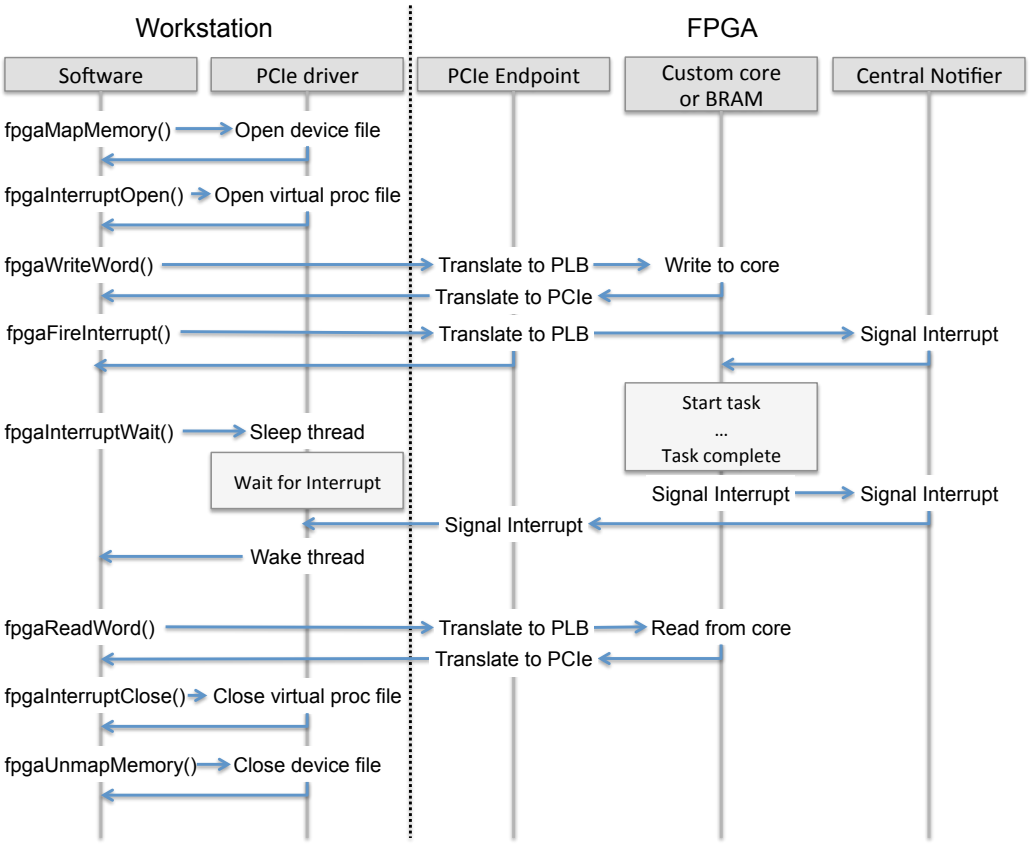


Figure 2.3. Example usage of RIFFA from a user application.

translate IP core accesses using PLB addresses to workstation PCIe memory accesses using workstation PCIe addressing. The PCIe-to-IPIF BARs work in exactly the opposite direction. These BARs support writing to the FPGA IP cores and receiving responses. This core is also configured to send MSI style interrupts to the Linux device driver. Xilinx provides the source for this core with its developer tools (a version of which is free). The core we use in our designs is the Xilinx plbv46_pcie ver. 3.0.0.a.

The DMA Controller is also a Xilinx IP core with similar no-fee licensing. It provides DMA transfer capabilities over the PLB bus. We configure this core to have a FIFO depth of 48 along with a read and write PLB burst size of 16. This core fires interrupts upon completion of DMA transfers. The core we use in our designs is the xps_central_dma ver. 2.0.1.b. Xilinx also provides the source for this core.

The Central Notifier is the heart of the RIFFA framework. It aggregates interrupt requests from IP cores and sends them to the PCIe Endpoint. It also handles DMA and interrupt requests using PLB mapped registers. The register space is partitioned into blocks which support DMA transfers and interrupts for the 16 possible IP core channels. Writes to registers in each block initiate a DMA transfer by setting the source address, destination address, transfer length, and interrupt flag. Upon receiving the length value, the Central Notifier queues a DMA transfer request into a FIFO to be issued to the DMA Controller. This reduces resources by requiring only one DMA Controller and leverages PLB arbitration to avoid request collisions. After the DMA transfer is complete, an interrupt/doorbell may be fired to the appropriate target (workstation or core) if the interrupt flag was set. Interrupts/doorbells may also be initiated by setting the interrupt flag and requesting a transfer length of zero.

The Central Notifier is also responsible for initializing the PCIe Endpoint. This is accomplished by writing to PCIe Endpoint registers over the PLB. The IPIF-to-PCIe BARs cannot be fully configured until the workstation boots and the Linux kernel assigns an address range. This value may change between reboots. Thus after each boot the driver transmits the kernel assigned address to the Central Notifier which then finishes the IPIF-to-PCIe BAR configuration.

The interface to custom IP cores is comprised of doorbell in, interrupt out, and DMA request/acknowledgment signals. The Central Notifier provides the interrupt signals for up to 16 different applications. DMA signaling is handled by the DMA Request core. This core exports a simplified set of signals for requesting DMA transfers and receiving completion information using a simple assert and pulse interface. The combined signals present an interface that simplifies the task of receiving events from software and responding with data. All interface functionality is also accessible via reads/writes from the PLB. Thus on-chip processors or any PLB master can make use of

RIFFA as well.

If a core requires parameter data, currently, it must allocate BRAM or memory channel cores on the PLB. Software on the workstation can then write or DMA transfer parameter data to the BRAM using the software interface. We intend to integrate support for temporary parameter data into future versions of RIFFA.

Performance

High bandwidth and low latency are among the criteria for this communications framework. We have profiled RIFFA running on Fedora 12 and Ubuntu 10.04 (Linux kernels 2.6.31-32). The FPGA designs were implemented using ISE and XPS 11.5 on a Xilinx ML506 board with a Virtex 5 XC5VSX50T running at 125 MHz. The ML506 board supports a single lane PCIe Gen 1 connection and was connected to a Dell Optiplex 745. The Dell has dual core Intel 2.4 GHz processors and 4 GB of RAM.

Latency times and bandwidths of key operations are listed in Table 2.2. Latencies were measured using cycles counted on the FPGA. The interrupt latency is the time from the FPGA signaling of an interrupt until the Linux device driver receives it. The read latency measures the round trip time of a request from the workstation to BRAM and the returned response (over the PCIe bus). Compared to the alternative freely available framework [25], the round trip latency is 36 times faster. The time to resume a user thread after it has been woken by an interrupt is the only latency that stands out. At 10.4 μs it represents the longest delay and is wholly dependent on the Linux kernel implementation.

The bandwidth measurements are for a single direction transfer between BRAM on the PLB and CPU main memory. We tested using DMA transfer sizes of 8 KB - 256 KB, by a factor of 2. The bandwidth in the direction of the FPGA to the workstation is sustained at 72% of the theoretical maximum for a single lane PCIe Gen 1 channel. The bandwidth is consistent across this range largely due to DMA pipeline depth. However

Table 2.2. RIFFA 1.0 key latencies and bandwidths.

Description	**Value**	
FPGA to PC interrupt time	3 μs	$\pm$ 0.06
PC read from FPGA round trip time	1.8 μs	$\pm$ 0.09
PC thread wake after interrupt time	10.4 μs	$\pm$ 1.16
FPGA to PC bandwidth	181 MB/s	$\pm$ 3.14
PC to FPGA bandwidth	25 MB/s	$\pm$ 1.22
Theoretical max 1x PCIe Gen 1 bandwidth	250 MB/s	

the bandwidth in the opposite direction is comparatively quite poor. We are currently investigating this bottleneck.

Despite the relatively poor performance of the workstation to FPGA bandwidth, we feel the PCIe based connection is superior to the Ethernet connection used in [25]. Newer FPGAs support additional PCIe lanes which will increase bandwidth further. Even with a single lane, the maximum sustained transfer rate is 1.5 times higher than what is possible over Gigabit Ethernet.

We did not test performance between off chip FPGA DRAM and CPU main memory as there are many configuration variables that affect the performance of off chip DRAM that do not affect the PCIe link.

Resource usage for RIFFA is listed in Table 2.3. By far the largest utilization is from the PCIe Endpoint core. This high utilization is specific to the Virtex 5 family of FPGAs. With newer Virtex 6 and Spartan 6 FPGAs, the slice register and slice LUT utilization is considerably lower due to additional PCIe interface hard macros. Virtex 6 FPGAs use only 2505 slice registers and 3763 slice LUTs for the same core. Similarly, the Spartan 6 family uses only 2208 slice registers and 2868 slice LUTs. This is consistent with the trend of increasing PCIe adoption for FPGA connectivity.

Table 2.3. RIFFA 1.0 resource utilization.

Core Name	Slice Regs	Slice LUTs	BRAMs	DSP48Es
Central Notifier	1051	1080	2	0
PCIe Endpoint	7899	8741	10	0
DMA Controller	577	782	0	0
DMA Request	245	215	0	0

2.4 RIFFA 2.0

The second generation of RIFFA, RIFFA 2.0 [39] is based only on the Xilinx provided PCIe Endpoint. This core is used to drive the gigabit transceivers for the PCIe link. It targets Spartan 6, Virtex 6, and 7 Series Xilinx devices and supports multiple FPGA per host PC. There is no dependence on the Xilinx Embedded Development Kit. Users can create designs using Xilinx ISE or Vivado Design Suites. It also represents a significant improvement in bandwidth.

2.4.1 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

Table 2.4. RIFFA 2.0 software (C/C++) interface.

Function Name & Description
`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.

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 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.

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 2.4 (for the C/C++ languages). We omit the Java and Python interfaces 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.

Figure 2.4 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.

Table 2.5. RIFFA 2.0 hardware interface.

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.

```
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);
```

Figure 2.4. RIFFA 2.0 software example in C.

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 2.5. 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 maintain 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 2.4.1, `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 2.5 shows a Verilog example matching the C example code from Figure 2.4. In this example, the user core receives data from the software thread, counts the number 4-byte words received, and then returns the count.

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

```
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);
```

Figure 2.5. RIFFA 2.0 hardware example in Verilog.

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.

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.

2.4.2 Architecture

On the FPGA, the RIFFA 2.0 architecture is a bus master DMA design connected to a Xilinx Integrated Block for PCI Express (Xilinx PCIe Endpoint) core (see Figure 2.6). 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 2.4.1). 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

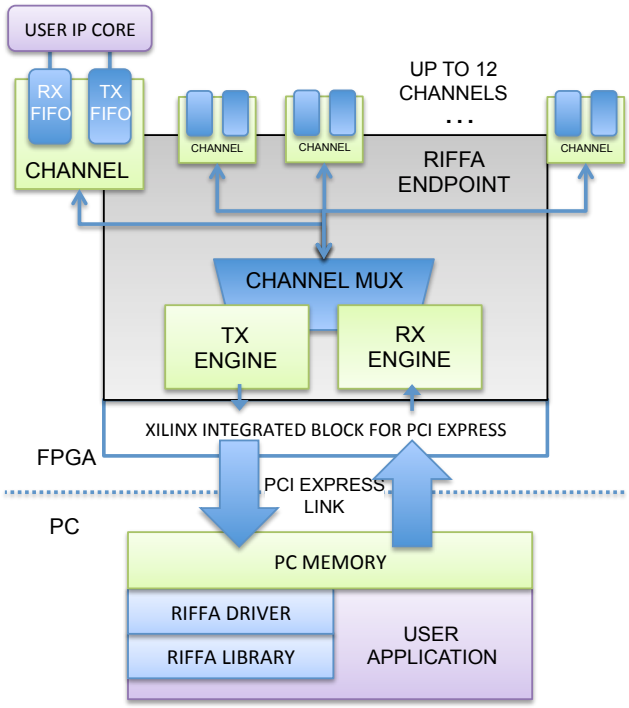


Figure 2.6. RIFFA 2.0 architecture.

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

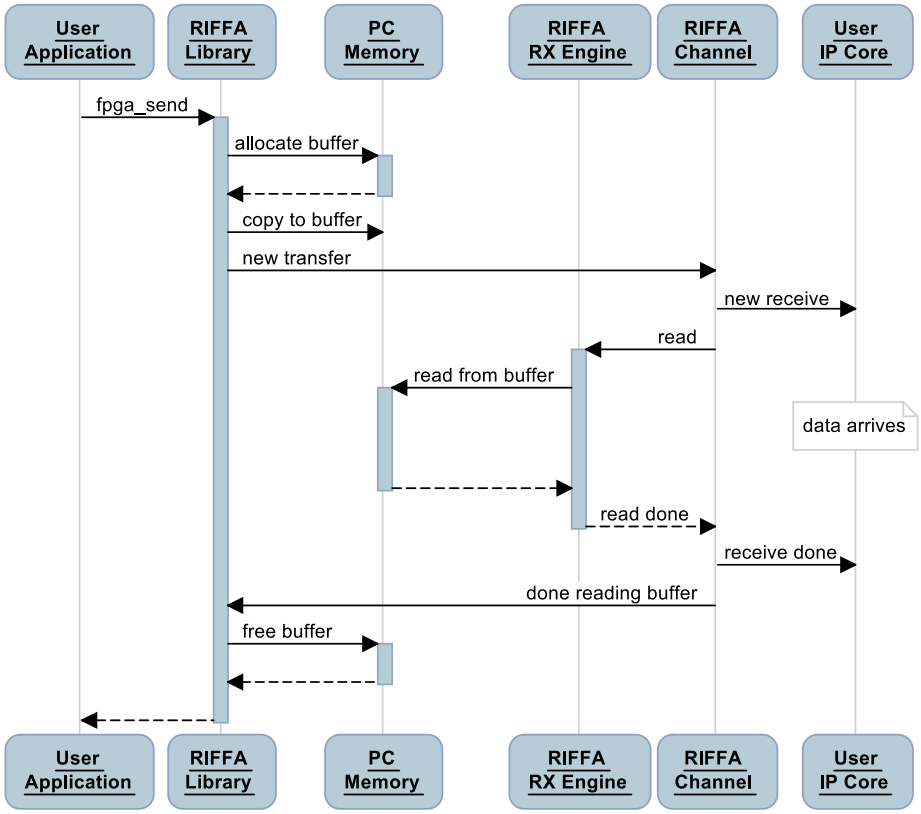


Figure 2.7. Downstream transfer sequence diagram.

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 2.4.1. 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.

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

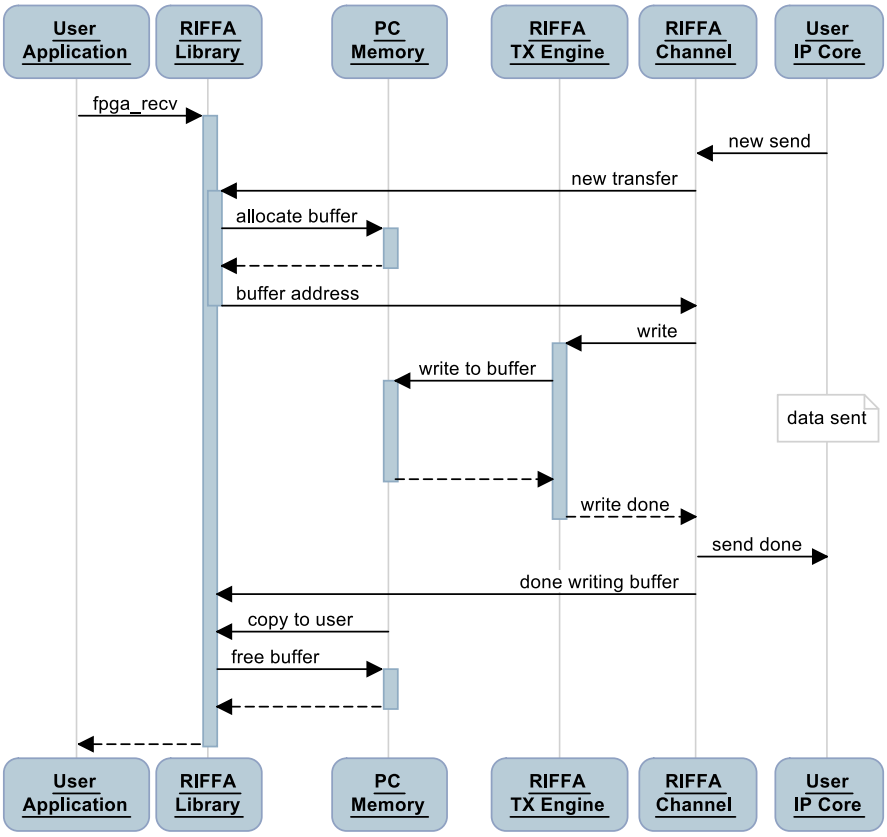


Figure 2.8. Upstream transfer sequence diagram.

data is to be transferred so that both are immediately aware of when the transfer ends.

Data Transfers

A sequence diagram for a downstream transfer is shown in Figure 2.7. 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 2.8. 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 2.7 and 2.8 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.

2.4.3 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 2.5. The software was operationally similar to the example listed in Figure 2.4.

Latency times of key operations are listed in Table 2.6. 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 2.9. 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 2.9 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 negligible 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.

Table 2.6. 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

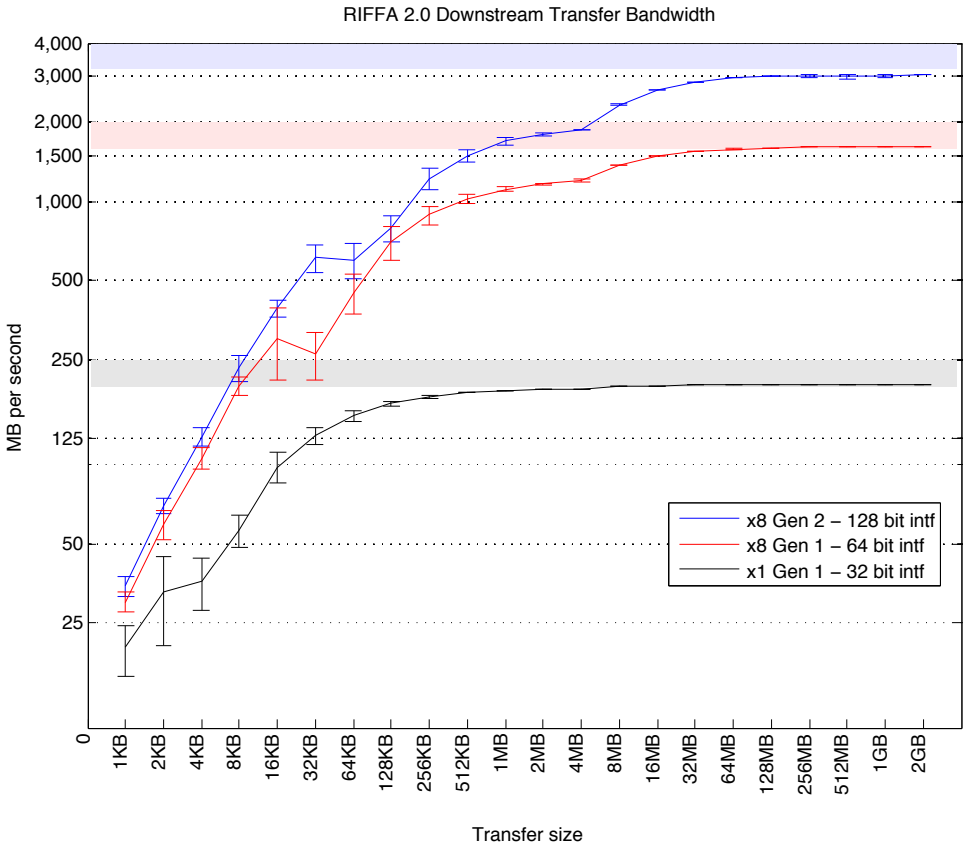


Figure 2.9. Downstream transfer bandwidths as a function of transfer size. Upstream bandwidths are nearly identical.

While not shown on Figure 2.9, 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 2.7. The cost for each additional channel is also listed. Resource values are from 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.

Factors Affecting Performance

Many factors go into attaining maximum throughput. There is enough confusion on the topic that Xilinx has published a whitepaper [28]. 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.

As Figure 2.9 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

Table 2.7. 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

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.

2.5 Conclusion

We have presented RIFFA, a reusable integration framework for FPGA accelerators. RIFFA 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 to further the growth of FPGA accelerated applications. RIFFA can be downloaded from http://cseweb.ucsd.edu/~mdjacobs.

Acknowledgment

This chapter contains material as it appears in Field-Programmable Custom Computing Machines (FCCM), 2012. It also contains material as it appears in Field Programmable Logic and Applications (FPL), 2013. The work described in this chapter is a collaboration with Yoav Freund. The dissertation author was the primary investigator and author of these papers.

Part II

Smart Frame Grabber Applications

Chapter 3

FPGA Accelerated Skin Color Detection

3.1 Introduction

Color detection is widely used in many computer vision applications. Tracking, segmentation, and even object detection can be performed by classifying pixel color. For some applications, specifying a fixed value or range of color in a color space provides sufficient detection accuracy. However, changes in lighting can often defeat such simple systems. Additionally, it can be difficult to identify the exact color values that match the target without also including similar non-target colors.

For this class of problem, it is often useful to rely on statistical methods instead of hand tuned heuristics. Machine learning algorithms can identify the correct colors given a set of positive an negative examples with higher accuracy than a human. Moreover, if the examples include varying lighting conditions the resulting detector can be more robust.

This paper describes our work in building a real-time skin color detector. We have adopted a machine learning approach for the task of skin color classification. Human skin color has many natural variations. It also looks very similar to other colors found in nature such as plant bark, wood, and sand. This makes it difficult to identify in real world

environments without also including non-skin pixels. Training a classifier with positive and negative examples can help identify the boundaries between skin and non-skin colors.

Our work in skin color detection also has a real-time component. It must classify all the pixels in each frame of video at camera rate with less than 100 μs of latency. Skin color classification is just a piece of a larger human-computer interaction (HCI) application. It therefore must execute with less latency than is tolerable for the entire application. To achieve this goal, we accelerate the detection on a FPGA.

The rest of this paper is organized as follows. We discuss the design and architecture of our skin color detector in Section 3.2. This is followed by our experimental results in Section 3.2. We close with conclusions.

3.2 Design and Architecture

Our application is partitioned between a CPU and a FPGA. The FPGA is responsible for capturing video frames from an attached camera, classifying each pixel at camera rate, and outputting the detections to the CPU. The CPU receives pixel classification data every frame and uses it to update the HCI application. The HCI application uses skin color information to identify regions within the frame that are likely to contain a human hand, head, or other uncovered body part.

We used the AdaBoost [27] algorithm to train our classifier, using only HSV color space pixel values. To gather training data, we collected videos of humans in the environments we expected to encounter during classification. Each video provided us with several thousand examples of positive and negative pixels. To improve classifier robustness, we included videos at different times of day. The natural lighting varies throughout the day and has a significant effect on image color values.

Boosting provided us with an alternating decision tree full of HSV color space features. Evaluating the alternating decision tree in software is trivial. However, evaluat-

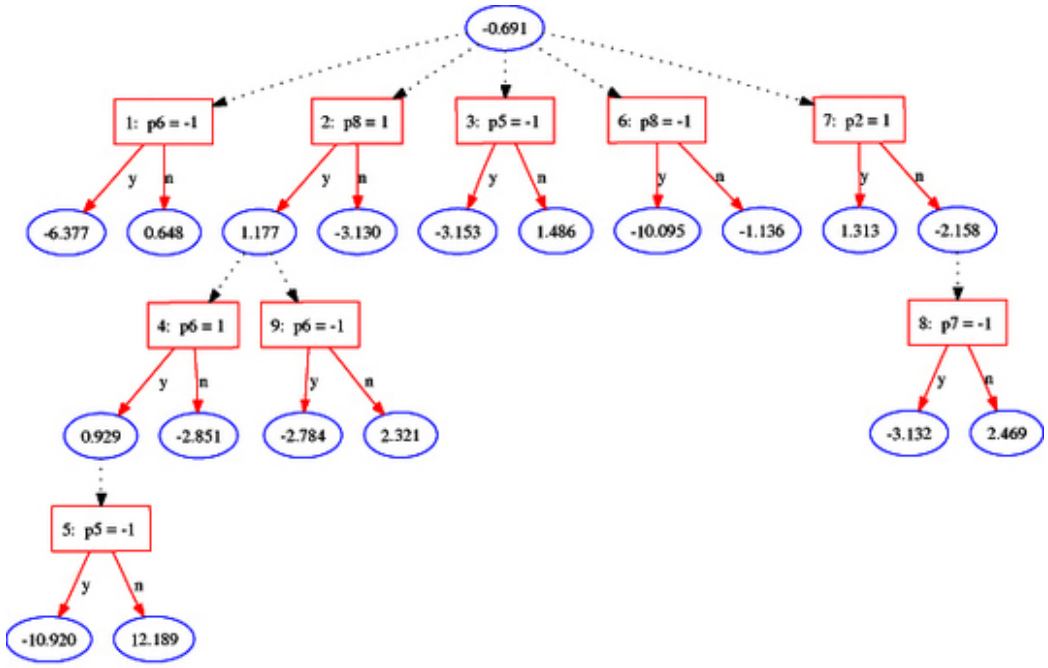


Figure 3.1. Alternating decision tree example.

ing each pixel in a video frame on a computer will not meet our real-time requirements. It therefore must run on the FPGA.

Implementing an alternating decision tree classifier in hardware is non-trivial. Unlike a simple decision tree which traces a single path through the tree, alternating decision trees can generate multiple parallel traces through a tree. This is because the outcome of evaluating a tree node can lead to multiple child nodes, not just one. The alternating decision tree example in Figure 3.1 shows how traversing the path labeled "y" from node 2 leads to simultaneous evaluation of nodes 4 and 9.

Evaluating each path in sequence is too slow. We therefore evaluate the paths in parallel, using a pipelined approach. Every cycle each node in the decision tree is evaluated for the current pixel. This produces a bitmap of 0/1 values encoding the true/false outcome of each node. Each node is only a numeric comparison to the HSV pixel values so the FPGA only needs n comparators, where n is the number of nodes in

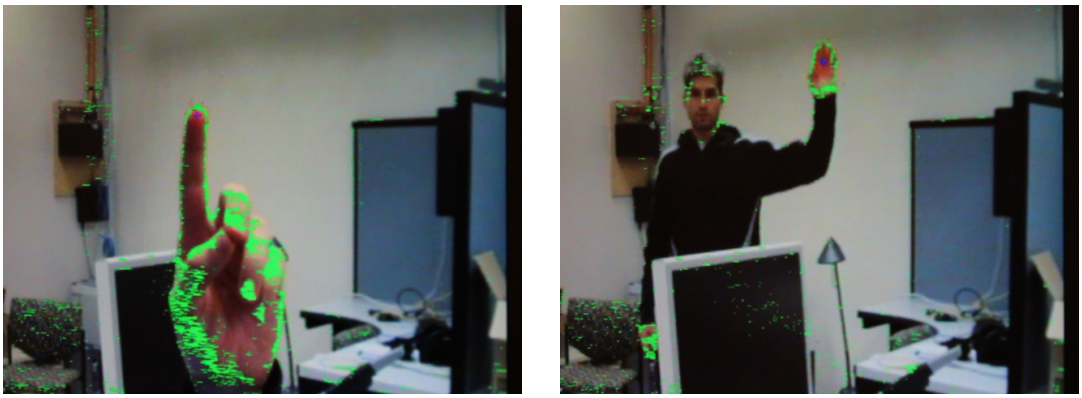
the tree. After the bitmap is generated, it is pushed through a pipeline that evaluates a portion of the results and incrementally builds a classification score.

The classification score is constructed by conditionally updating an intermediate score value every cycle. Because the node score values are known, the aggregate leaf node scores can be calculate ahead of time. This means only the leaf nodes need to be evaluated at each stage. The intermediate score is updated with the leaf node values only if the path to the leaf node was actually traversed. To implement this conditional addition, we simply use bit masks on the bitmap vector to identify if a path should be traversed. This process effectively evaluates all possible paths in the tree over several cycles and conditionally adds the leaf node values for each path to the score. At the final stage of the pipeline, the score represents the collective score of every path that is traced by the pixel value through the tree.

From a resource perspective this approach is very efficient. There is no pixel buffering of any kind. The number of comparators required is $O(n)$ and the number of conditional adders is $O(m)$ where n is the number of nodes in the tree and m is the number of leaf nodes in the tree. Implementing the conditional aspect of addition is only a bitwise AND operation over p bits, where p is the height of the tree. In practice this uses very few FPGA resources and can be stretched over many stages to accommodate clock frequency.

3.3 Performance

We implemented our algorithm on a Xilinx Virtex 5 ML506 development board. The board was connected directly to a camera and also to a monitor. The camera video was captured, classified, annotated, and then outputted to the attached monitor. Annotations replaced skin colored pixels with bright green pixels. This was done to easily identify the regions. The system was able to process 640×480 resolution video at



(a) **(b)**

Figure 3.2. Image frames of skin detector evaluating live video.

60 frames per second (camera rate) with no perceptible latency. Figures 3.2a and 3.2b show annotated outputs of the detector running on live camera video.

Despite training in multiple environments with different lighting. The detector still misses many pixels that are actual human skin. But the precision is reasonably high at the low recall it provides. In practice this can be accommodated with a lower application threshold for detected skin colored pixels.

3.4 Conclusion

We have described a FPGA accelerated skin color detector that uses an AdaBoost trained alternating decision tree classifier. It classifies human skin colored pixels in VGA resolution video at 60 frames per second with no perceptible latency. The design is resource efficient and can be pipelined to support higher resolutions at faster clock rates.

Chapter 4

FPGA Coprocessor for Particle Filter Tracking

4.1 Introduction

Realtime computer vision applications are latency sensitive and frequently require processing of high bandwidth data. To demonstrate the practicality of the RIFFA framework [38], we chose to accelerate a particle filter tracking application using VGA video to track multiple targets. Particle filters have been accelerated using FPGAs in standalone [17, 5] and integrated [3] designs. Our implementation however, supports tracking multiple targets at 60 Hz (camera rate).

Particle filters are a class of tracking algorithms based on Bayesian filtering [2]. Unlike the Kalman filter, particle filters can handle observation and dynamic models of targets that are not strictly Gaussian. This is accomplished by sampling from a proposal distribution because computing the Bayes optimal solution in closed form is not possible. The particle filtering algorithm we chose to accelerate is based on the NormalHedge online learning algorithm [14]. Unlike traditional particle filtering approaches, the NormalHedge tracking algorithm does not require a generative model. Instead, particles serve as a sequence of states that help explain the observations seen so far. States are guaranteed to be within a bounded margin of the optimal state. Each particle is weighted

in proportional to how likely it represents the true state of the object being tracked.

This accelerated tracking application is a demonstrable end-to-end application that uses FPGA acceleration to support multiple targets at 60 Hz. It is built using the RIFFA framework and is partitioned across a CPU and FPGA. This represents the chief contribution of this paper.

The rest of the paper describes in detail the architecture and performance of the accelerated tracker application. The algorithm is described in 4.2. The architecture is explained in Section 4.3. Experimental results can be found in Section 4.4. We conclude with a discussion of future work in Section 4.5.

4.2 Algorithm

The algorithm is listed in Algorithm 1. It begins by registering the location of the target to be tracked. In our system this is done manually, though nothing prevents us from changing this to be automatic. Once located, a template of the target is saved (i.e. the pixels representing a window containing the object). This template will be used by the *Loss* function during tracking. Once registered the target (and its location) can be tracked from frame to frame.

The algorithm follows the same pattern as particle filters, with some minor changes. The *Loss* function compares the target template with pixels in the current frame surrounding the particle. We calculate the loss as the sum of the L1 differences between pixel values in the R, G, and B channels. The *Resample* function resamples only particles with non-positive regret instead of all particles. Regret measures the amount of aggregated loss sustained by that particle from the optimal location over time. Particles are then reweighted according to a balancing constant, c. At each iteration, this constant is found (using a binary search) such that it solves $\frac{1}{N}\sum_{i=1}^{N} exp(\frac{R_{i,t}^2}{2c}) = exp$. This will bound the amount of error each particle can sustain. Lastly, the $UpdateState$ behaves

Algorithm 1. NormalHedge Tracking Algorithm

$A = \{x_{1,1}, x_{2,1}, ..., x_{N,1}\}$, $x_{i,1}$ ▷ Drawn randomly
$R_{i,0} = 0$; $w_{i,0} = 1/N \; \forall_i$
for t = 1, 2, ... **do**
    $l_{i,t} = Loss(x_{i,t})$ ▷ Loss for each particle
    $l_{A,t} = \sum_{i=1}^{N} w_{i,t-1} l_{i,t}$ ▷ Average loss
    $R_{i,t} = (1-\alpha)R_{i,t-1} + (l_{A,t} - l_{i,t})$ ▷ Update regrets
    $B = \{i : R_{i,t} \leq 0\}$ ▷ Bad particles
    $A = AnB$ ▷ Remove bad
    $A = A \cup Resample(A, B)$ ▷ Resample
    $c = ComputeC(A)$
    $w_{i,t} \propto \frac{R_{i,t}}{c} exp(\frac{R_{i,t}^2}{2c})$ ▷ Re-weight
    $x_{A,t} = \sum_{i=1}^{N} w_{i,t} x_{i,t}$ ▷ Estimate location
    $x_{i,t+1} = UpdateState(x_{i,t}) \; \forall i$ ▷ Update state
end for

the same as with most other particle filters by updating each particle according to its dynamics model. Our dynamics model is a simple velocity based model.

4.3 Architecture

It may not be immediately clear from the algorithm, but the overwhelming majority of time is spent calculating the loss values. Doing so requires comparing hundreds of pixels for each particle. In most particle filtering algorithms, the number of particles used determines the program's run time. The more particles used however, the better the tracking performance. In light of this, we partitioned the application so that the repetitive, high bandwidth loss calculation was performed by cores on the FPGA. The rest of the algorithm, involving data structure manipulation and search, was implemented in software. This approach is much faster to implement when compared to building a complete standalone design in hardware. It also exemplifies how using RIFFA can exploit the strengths of both the FPGA and workstation platforms.

The architecture for the tracker is depicted in Figure 4.1. The algorithm runs in software on the workstation. For each iteration of the algorithm, the software calls the

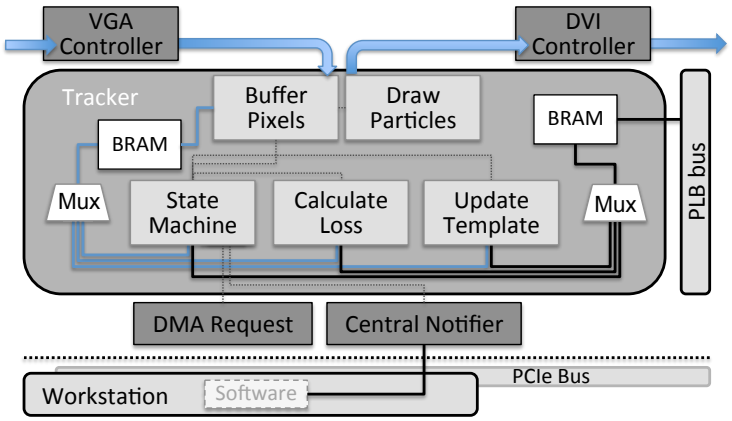


Figure 4.1. Tracker core architecture. The State Machine module interfaces with the DMA Request and Central Notifier cores. Software on the workstation initiates processing via interrupts.

Tracker core to calculate the *Loss* function. This requires writing function parameters to BRAM and then interrupting the Tracker core. The Tracker core is driven by the video clock and receives video data directly from the camera. When triggered, data entering the core is processed and loss values are DMA transferred back to the workstation. Additionally, output video is annotated with target tracking information and sent to the DVI Controller. The processing runs as the pixels stream through the system in real time. Once the response data has been received by the workstation, the Tracker core will interrupt the waiting software thread to continue running the algorithm. The regret values are updated, particles are resampled and reweighted, and a new location is estimated.

The Tracker core can operate at 4 different scales: 1, 2, 4, and 8. The processing is coordinated by the State Machine module. When processing a frame, one line of video data is buffered by the Buffer Pixels module. If the runtime parameters indicate the target template should be updated, then the Update Template module uses the line pixels to update the template. Otherwise, for each particle, the line of pixel data is compared to the corresponding line in the stored template and a partial loss value is computed. Particles can be located anywhere around the target location, so the Calculate Loss module must

determine which particles intersect with the saved line. The Tracker core must handle the case where every particle intersects each line. After score calculation the line buffer is overwritten with particle locations for the following line. In the next iteration, the Buffer Pixels module will read the locations when saving new data, then pass the location data to the Draw Particles module so it can annotate the outgoing video. A two line FIFO buffer is used so that the line processing can overlap with the line saving. The line processing must complete within a single line of video. Once all the lines are processed, the State Machine module DMA transfers loss values to the workstation and notifies the application via an interrupt.

In addition to the Tracker core, there are other cores that support video processing. These cores handle the formatting of input from a VGA camera and format output video for a DVI monitor. Additionally, a Frame Capture core is used to capture video frames so the workstation can render video.

The Tracker core reads a number of software defined runtime parameters upon each invocation. These include: the target's x, y coordinate location in the video frame, whether to update the template or calculate loss values, the video scale factor, the maximum particle spread, the template size, and the number of particles. Additionally, the x, y location of each particle is provided. The response data from the Tracker core includes the video frame number and the loss values for each particle.

4.4 Performance

The software portion of the NormalHedge tracker is written in C and makes use of multiple threads. We used the OpenCV library for rendering frames on the workstation. The FPGA cores were implemented using ISE and XPS 11.5, on a Xilinx ML506 with a Virtex 5 XC5VSX50T running at 125 MHz. We tested using a Dell Optiplex 745 with dual core Intel 2.4 GHz processors and 4 GB of RAM.

Table 4.1. Tracker and related video core resource utilization.

Core Name	Slice Regs	Slice LUTs	BRAMs	DSP48Es
Tracker	767	2607	1	2
VGA-DVI Controller	183	325	0	0
Frame Capture	144	205	1	0

We were able to fit 6 Tracker cores, the additional video processing cores, and RIFFA on our Virtex 5. The Tracker core is able to process VGA (640x480) video at over 400 frames per second. In our experiments we tested the system with live VGA data at 60 Hz. This resulted in a 25.125 MHz video clock frequency. The resource utilizations for the Tracker and video processing cores are listed in Table 4.1.

To test overall performance, we ran several experiments tracking individual hands, fingers, etc. Each of the targets were tracked with 100 particles. We found the tracking to perform adequately when the user was not trying to defeat the system. Normal movements can be tracked, but are lost if the motion is too fast or the target is occluded. We found that background colors in the template tend to confuse the algorithm when the target moves to a location with a very different background color. The accelerated application was able to track at the 60 Hz camera rate using only 25-40% of the workstation CPU capacity. Unaccelerated, the application was only able to operate at 2 Hz, using 100% of the CPUs. This represents a demonstrable 30x performance increase while using just a fraction of the CPU processing power. A video of the tracker is available at http://photobucket.com/submissionvideos.

4.5 Conclusion

We presented a FPGA accelerated application of tracking using the RIFFA framework. The tracker uses a particle filter approach to search for targets in the input frame. The system is capable of tracking 6 independent targets at a rate of 60 Hz. By using

RIFFA we can partition the application across the CPU and FPGA, leveraging both platforms' resources. This shows how this approach can improve performance to state of the art levels.

Chapter 5

FPGA Accelerated Face Detection

5.1 Introduction

Face detection is a popular task to accelerate as it is frequently the first step in a larger application. It is also difficult to perform in realtime at even modest resolutions. We selected the well known Viola and Jones face detector [58] for acceleration due to its wide spread use and robust performance. Prior work exists for Viola and Jones detectors on FPGAs [61, 44]. Most are standalone designs and are not capable of running with live video. Additionally, many only implement the basic feature calculation portion of the algorithm and are not designed to function as a complete detector. Others still, are complete, but don't support detection at VGA resolutions or support multiple scale detection. Implementing all required aspects of the algorithm to support a demonstrable system can be challenging. Some research however, has been complete in this sense [18] and we compare our work most closely to theirs.

This accelerated face detection application is a demonstrable end-to-end application that uses FPGA acceleration to support realtime face detection. It is built using the RIFFA framework and is partitioned across a CPU and FPGA. This represents the chief contribution of this paper.

The rest of the paper describes in detail the architecture and performance of

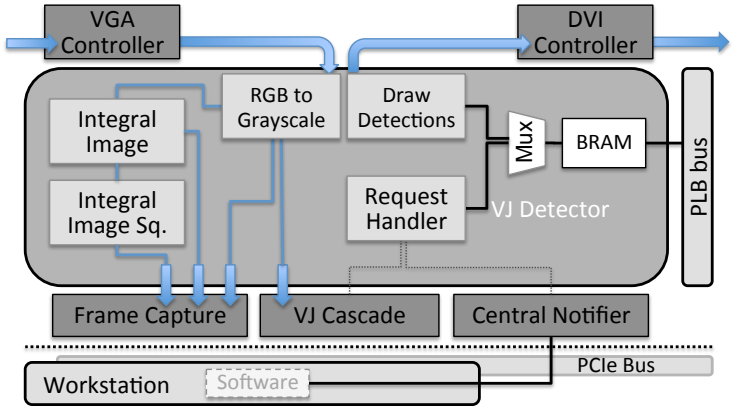


Figure 5.1. Architecture of the VJ Detector core. The Request Handler module interfaces with the Central Notifier and VJ Cascade cores. Software on the workstation initiates processing via interrupts.

the accelerated face detection application. The algorithm is described in 5.2. The architecture is explained in Section 5.3. Experimental results can be found in Section 5.4. We conclude with a discussion of future work in Section 5.6.

5.2 Algorithm

The Viola and Jones detection algorithm operates on grayscale data and uses pairs and triples of adjacent rectangles as features. The sums of the pixel values in each rectangle are weighted and summed to form a feature value. Feature values are extracted from candidate locations within an image frame. The set of candidate locations is an overlapping set of square regions, typically extracted at many scales. For each candidate location, features are calculated sequentially according to a cascade. The cascade is an ordering of features, grouped by efficacy, into stages such that the features in the earlier stages have more discriminative power. For each stage in the cascade, the constituent feature values are summed and compared against the stage threshold. Sums less than the threshold indicate low likelihood of containing a face, and the candidate is abandoned. Sums exceeding the threshold allow the candidate to continue to the next stage. If a

candidate exceeds the thresholds for all stages in the cascade, it is labeled as containing a face.

Because summing pixel values is expensive, the algorithm converts grayscale images into summed area tables called integral images. Pixel values in the integral image represent the sum of grayscale values above and to the left of the same pixel in the original image. This conversion enables feature values to be calculated in constant time. An additional image is also generated, the integral image squared. This is similar to the integral image, but is constructed using the square the the original pixel values. Using both images, the intensity variance can be calculated to normalize images across different lighting conditions.

All the features, weights, stages, and thresholds are learned using Boosting. The order of features in the cascade is designed so that candidate locations that do not contain a face are identified early in the cascade with as few features evaluated as possible. The number of features in the cascade depends on the training, but is on the order of a few thousand for detecting faces. The unscaled candidate location size for face detection is typically 20x20 pixels.

5.3 Architecture

Partitioning the Viola and Jones algorithm is challenging because performing any stage of the cascade requires building nearly all the functionality needed to perform the entire cascade. As a result, most FPGA accelerations of this algorithm are implemented completely on the FPGA. In our design, we implemented the entire detector cascade in software and the first few stages of the cascade on the FPGA. The partial FPGA cascade is a fully parallel design. This allows us to evaluate these stages at camera rate. Performing only a few stages of the cascade is useful because the results can be used as a filter for the full cascade, run in software. We used the trained face detector specification

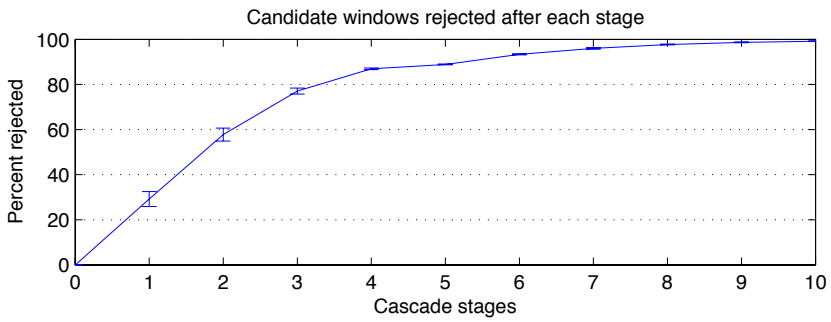


Figure 5.2. Cumulative percentage of candidate locations rejected after evaluating cascade stages. Only first 10 stages shown.

available in the OpenCV library as design parameters for both our software and hardware implementations.

The primary motivation for our design is illustrated by the graph in Figure 5.2. There are 22 stages and 2135 features in the entire cascade. But on average, we found that 87% of the candidate windows have been rejected by the end of stage 4. Only needing to run the cascade on 13% of the frame can greatly improve the runtime performance. Moreover, doing so only requires evaluating 79 features.

Diagrams of our hardware detector are shown in Figures 5.1 and 5.3. As with the tracker application, the algorithm runs in software on the workstation. For each iteration of the algorithm, the software requests frame data from the VJ Detector core. The request parameters are transferred to BRAM, then the core is interrupted. The core generates grayscale, integral image, and integral image squared frame data from the input frame. The integral image and integral image squared frames are represented using 32 bit integers and IEEE 754 floats respectively. This data is saved to the workstation using Frame Capture cores. The VJ Cascade is responsible for running feature extraction and stage thresholding for the implemented cascade stages. The output of the VJ Cascade is a bitmap. Each position represents a candidate location. If the value at a location is 1, the candidate has been rejected. If the value is 0, the candidate must be evaluated further.

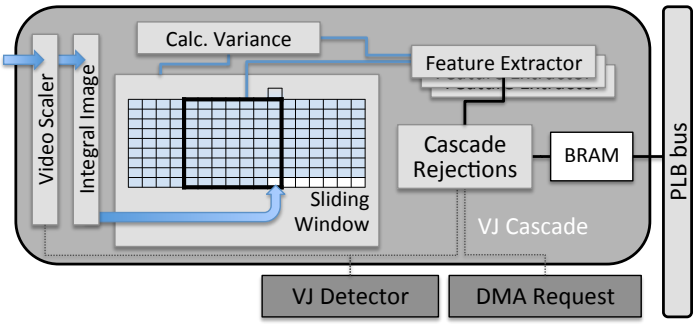


Figure 5.3. Architecture of the VJ Cascade core. The detections from the first few stages are saved as a binary bitmap by the Cascade Rejections module.

The processing runs as the pixels stream through the system in real time. Thus by the end of a frame all the response data has been DMA transferred to the workstation. The response is comprised of an integral image frame, an integral image squared frame, and a candidate rejection bitmap from each VJ Cascade core. Once the response data has been received by the workstation, the algorithm is notified via an interrupt and the entire cascade is run on the response data using the bitmaps as candidate filters.

The VJ Detector core is coordinated by the Request Handler module. It receives runtime parameters for scale, video capture type, and detection locations. The response data after each invocation are the video frame(s) and rejection bitmap(s). In addition to capturing frame data, the core annotates output video with detection information.

The VJ Cascade core generates a rejection bitmap every video frame. The Video Scaler module scales grayscale video according to the request value. It scales by subsampling streaming video and supports 18 scales. The data is then converted into integral image data and captured by the Sliding Window module. This module stores pixels in BRAM line buffers and systolically shifts pixels across lines (BRAMs) to produce a sliding window across the video frame. Each new pixel produces a new column of video data. The columns are aggregated using registers to provide random access to any pixel in the window. These registers are accessed by the Calc. Variance module

Table 5.1. Face detector core resource utilization.

Core Name	Slice Regs	Slice LUTs	BRAMs	DSP48Es
VJ Detector	716	882	2	1
2-stage VJ Cascade	3529	7267	23	107
3-stage VJ Cascade	6248	12,698	23	195
4-stage VJ Cascade	9552	26,263	23	287
5-stage VJ Cascade	11,708	49,437	23	287

to calculate the variance for the current window. The Calc. Variance module performs the same normalization as is done with the integral image and integral image squared frames. The window registers are also accessed by each Feature Extractor module. They calculate a new feature value each cycle, completely in parallel. The feature values are then summed and compared to their stage thresholds in the Cascade Rejections module. The outcome is saved to as a bitmap and DMA transferred it to the workstation.

5.4 Performance

The software portion of the face detector is written in C using multiple threads. The FPGA cores were implemented on a Virtex 5 XC5VSX50T running at 125 MHz, using ISE and XPS 11.5. Again we tested using a Dell Optiplex 745 with dual core Intel 2.4 GHz processors and 4 GB of RAM.

We fit two designs on our Virtex 5. One with a 3-stage VJ Cascade core and one with dual 2-stage VJ Cascade cores. We also synthesized a design with quintuple 4-stage VJ Cascade cores for a Virtex 6 XC6VLX240T. Though the VJ Cascade core can support VGA (640x480) video up to 120 Hz, our tests were run with live VGA video at 60 Hz. The resource utilizations for the face detector cores with various stages are listed in Table 5.1.

Our experiments use the same 20x20 pixel candidate window size as the OpenCV

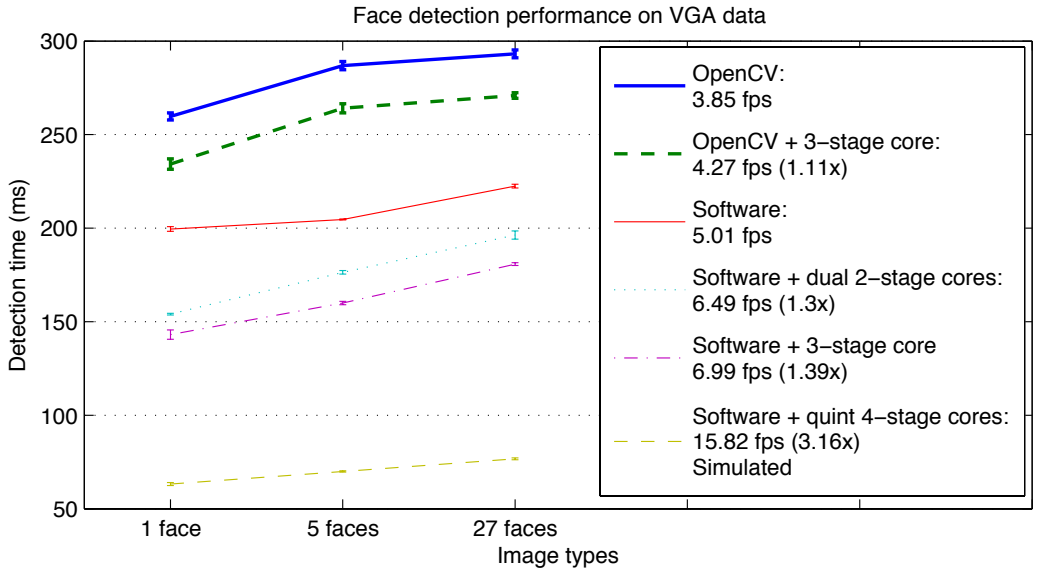


Figure 5.4. Face detection times on VGA video. Speed up over equivalent software only version is listed in parenthesis.

implementation. A step size and scale factor of 1.2 was used. This corresponds to 888,634 candidate windows over 18 scales on VGA video. In all tests, the highest resolution scales were accelerated. As the detection time can vary across images, we tested using images with 1, 5, and 27 faces. The performance results are shown in Figure 5.4. We were able to improve the detection frame rate over a software only version by 1.39× using the single 3-stage VJ Cascade design on our Virtex 5. Using a Virtex 6 however, our simulations project a 3.16× increase for a frame rate of 15.82 fps. Compared to similar work, we feel this performs at state of the art levels.

5.5 OpenCV Integration

In addition to implementing our own face detector, we integrated the FPGA accelerator into the OpenCV face detection library. OpenCV contains a highly optimized implementation of the Viola and Jones detector. Much of their performance comes from using a minimum step size of 2, adaptively skipping every other candidate location,

```
foreach (Window w in frame.windows) {
  bitmap = bitmaps[w.scale];
  if (bitmap != NULL) {
    if (bitmap[w.position])
      continue;
    else
      run_cascade_from_stage(n, regions, ii, iis);
  }
  else {
    run_cascade_from_stage(0, regions, ii, iis);
  }
}
```

Figure 5.5. Pseudo code representation of modified OpenCV cascade detection function.

skipping the lowest scale (original resolution), and filtering the image using edge detection heuristics.

Updating the OpenCV classifier routines to make use of the rejection bitmaps involved modifying the OpenCV function prototypes to accept a pointer to the bitmaps, passing along the pointer as necessary, and in the appropriate function, evaluating the value of the bitmap for the current candidate window before running the cascade. While this did involve modifying existing OpenCV source (mostly `cascadedetect.cpp`), it was the most efficient way to make use of the rejection bitmaps. A pseudo code representation of the modification is presented in Figure 5.5.

5.5.1 Performance

Our experiments use the same 20x20 pixel candidate window size as the OpenCV implementation and the same FPGA design as before. The performance results for the OpenCV integrated experiments are shown in Figure 5.6. We were able to improve the detection frame rate over the OpenCV software version by 1.1× using the single 3-stage VJ Cascade design on our Virtex 5. Our device limited us to a single scale. However we calculated a 4.1× increase for a frame rate of 15.82 fps on a Virtex 6 using 5 VJ Cascades, each configured with 4-stage features.

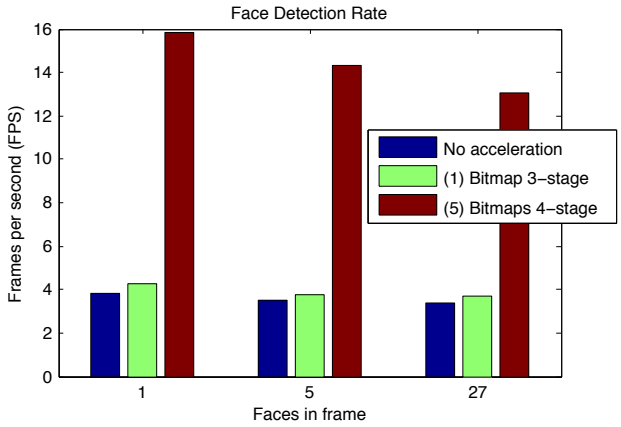


Figure 5.6. Face detection times on VGA video in frames per second.

Because of the optimizations in the OpenCV implementation, adding the FPGA rejection bitmap acceleration did not improve performance as much as in our software implementation. However, to our knowledge, this work represents the first FPGA accelerated face detector integrated into the OpenCV library.

5.6 Conclusion

We presented a FPGA accelerated application of face detection using the RIFFA framework. The face detector is based on the Viola and Jones detector using Haar features. The system is capable of demonstrating a 1.39× increase using a Virtex 5 FPGA over a highly optimized CPU implementation. Our simulations project a 3.16× increase when using a Virtex 6 FPGA. By using RIFFA we can partition the application across the CPU and FPGA, leveraging both platforms' resources. This shows how this approach can improve performance to state of the art levels.

Chapter 6

FPGA-GPU-CPU Heterogenous Architecture for Real-time Cardiac Physiological Optical Mapping

6.1 Introduction

Optical mapping technology has proven to be a useful tool to record and investigate the electrical activities in the heart [37][49]. Unlike other cardio-electrophysiology technologies, it does not physically interfere with the heart. It provides a dense spatial electrical activity map of the entire heart surface. Each pixel acts as a probe on that location of the heart. Variation in pixel intensity over time is proportional to the voltage at that location. Thus a 100×100 resolution video is equivalent to 10,000 conventional probes. This produces more accurate and comprehensive information than conventional electrode technologies.

The process of optical mapping involves processing video data to extract biological features such as depolarization, repolarization and activation time. The challenge in this process is primarily in the image conditioning. Raw video data contains appreciable sensor noise. Direct extraction of biological features from the raw data yields results too inaccurate for most medical use. Therefore, the process includes an image conditioning algorithm, which has been presented and validated by Sung *et. al* [56]. The effect of this

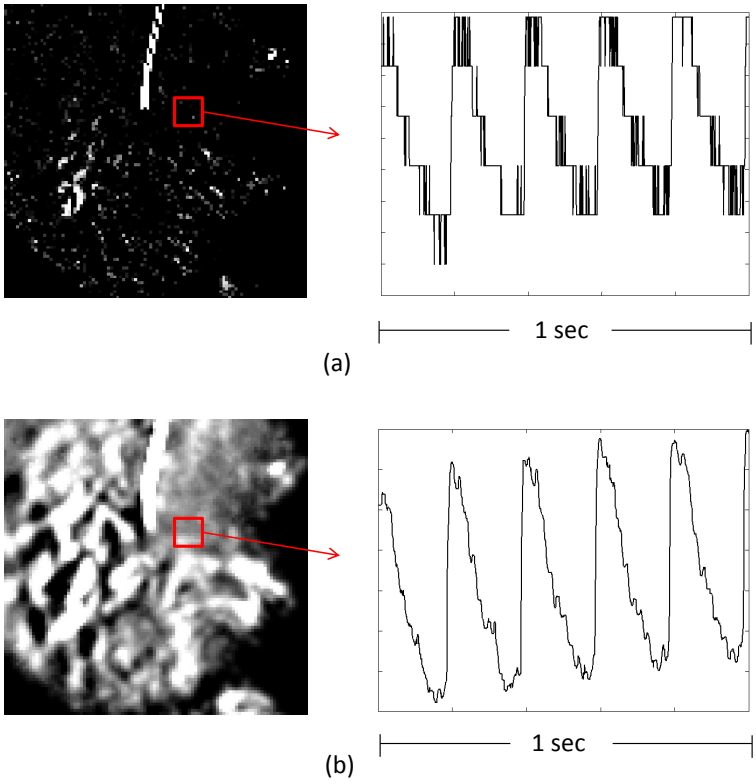


Figure 6.1. Image conditioning effect (left: the grayscale image of a random frame, right: the waveform of a random pixel over time). (a) before image conditioning. (b) after image conditioning.

image conditioning is shown in Figure 6.1.

Real-time optical mapping is useful and potentially necessary in a wide range of applications. One domain is real-time closed loop control systems. This includes dynamic clamp [11, 23], and the usage of tissue-level electrophysiological activity to prevent the onset of arrhythmia [24, 33]. These systems offer the unique ability to understand the heart dynamics by observing real-time stimulus/response mechanisms over a large area. Another domain of applications is immediate experimental feedback. The ability to see the optical mapping results during the experimental procedure can significantly reduce both the duration of the experiment and the required number of experiments.

Achieving real-time optical mapping is computationally challenging. The input data rate and the required accuracy for biological features results in a throughput on the order of $10,000$ fps. At such high throughput, a software implementation takes 39 mins to process just a second of data. Even a highly optimized GPU accelerated implementation can only reach 578 fps. A FPGA-only implementation is also infeasible due to the resources required for processing intermediate data arrays generated by the optical mapping algorithm.

In this paper, we propose a real-time FPGA-GPU-CPU heterogenous architecture for cardiac optical mapping that runs in real-time, capturing 100×100 pixels/frame at 1024 fps with only 1.86 seconds of end to end latency. Experimental parameters and data are based on the experiments by Sung *et. al* [56]. Our design has been implemented on an Intel workstation using an NVIDIA GPU and a Xilinx FPGA. The implementation is a fully functioning end to end system that can work in an operating room with a suitable camera.

The contributions of this paper are:

- A real-time optical mapping system using a FPGA-GPU-CPU heterogeneous architecture.
- An optical mapping partitioning analysis for heterogeneous accelerators.

The rest of the paper is organized as follows. We discuss related work in Section 6.2. In Section 6.3, we describe the optical mapping algorithm in detail. We discuss algorithm partitioning decisions in Section 6.4. We describe the design and implementation of the heterogenous architecture in Section 6.5. In Section 6.6, we present the experimental results and accuracy of our implementation. In Section 6.7, we conclude.

6.2 Related Work

The optical mapping process involves three types of computations: spatiotemporal image processing, spectral methods, and sliding-window filtering that can result in performance challenges. A variety of approaches have been proposed to accelerate image processing algorithms that have one or more of these computations. There are FPGA and GPU accelerated approaches for real-time spatiotemporal image processing [13] [50]. Govindaraju *et. al* have analyzed the GPU performance on spectral methods [29]. Pereira *et. al* have presented a study of accelerating spectral methods using FPGA and GPU [52]. Many sliding-window filtering applications have been presented in the past [26] [21]. None of the approaches described above combine all three of the computations as in the optical mapping algorithm.

Several FPGA-GPU-CPU heterogeneous acceleration systems have been proposed in recent years. Inta *et. al* have presented a general purpose FPGA-GPU-CPU heterogeneous desktop PC in [36]. They reported that an implementation of a normalized cross-correlation video matching algorithm using this heterogeneous system achieved 158 fps with 1024×768 pixels/frame. However, they ignored the throughput bottleneck of the PCIe which is critical in real-time implementations. Bauer *et. al* have proposed a real-time FPGA-GPU-CPU heterogeneous architecture for kernel SVM pedestrian detection [8]. However, instead of having spatiotemporal image processing and spectral methods (across frames), this application only has computations within individual frames.

We present a stage level algorithm partitioning according to the computational characteristics and data throughput. To the best of our knowledge, the system presented in this paper is the first implementation of a real-time optical mapping system on a heterogenous architecture.

6.3 Optical Mapping Algorithm

Figure 6.2 (a) depicts an overview of the algorithm. Video data is provided by a high frame rate camera. The input video data is zero score normalized to eliminate the effects of varying background intensities. After normalization, there are two major noise removing facilities: a phase correction spatial filter and a temporal median filter.

6.3.1 Normalization

Normalization is performed for each pixel in a temporal fashion, across frames. In our experiments the input video arrives at 1024 fps. Normalization is performed on each second of video, disjointly. To compute the normalization base value for pixel location, we find the weighted mean of the largest three values in the temporal array. We can then normalize each pixel in the frames using Equation 6.1 with the correspondent normalization base value.

$$normed.\ pixel = 100\frac{-(raw\ pixel - base\ val.)}{base\ val.} \tag{6.1}$$

6.3.2 Phase Correction Spatial Filter

The action potential is distributed as a waveform on the heart surface. Thus, if we merely apply a Gaussian spatial filter on the video data, we will lose the critical depolarization properties (the sharp edges of the waveform in cardiac physiology). Therefore, a phase correction algorithm needs to be applied to cause the pixels in the window to be in phase before the Gaussian spatial filter.

The phase correction spatial filter operates as a sliding window function across the entire frame, where each operation uses all frames across time (see Figure 6.2 (b)). Figure 6.2 (c) illustrates an example 5×5 Gaussian filter.

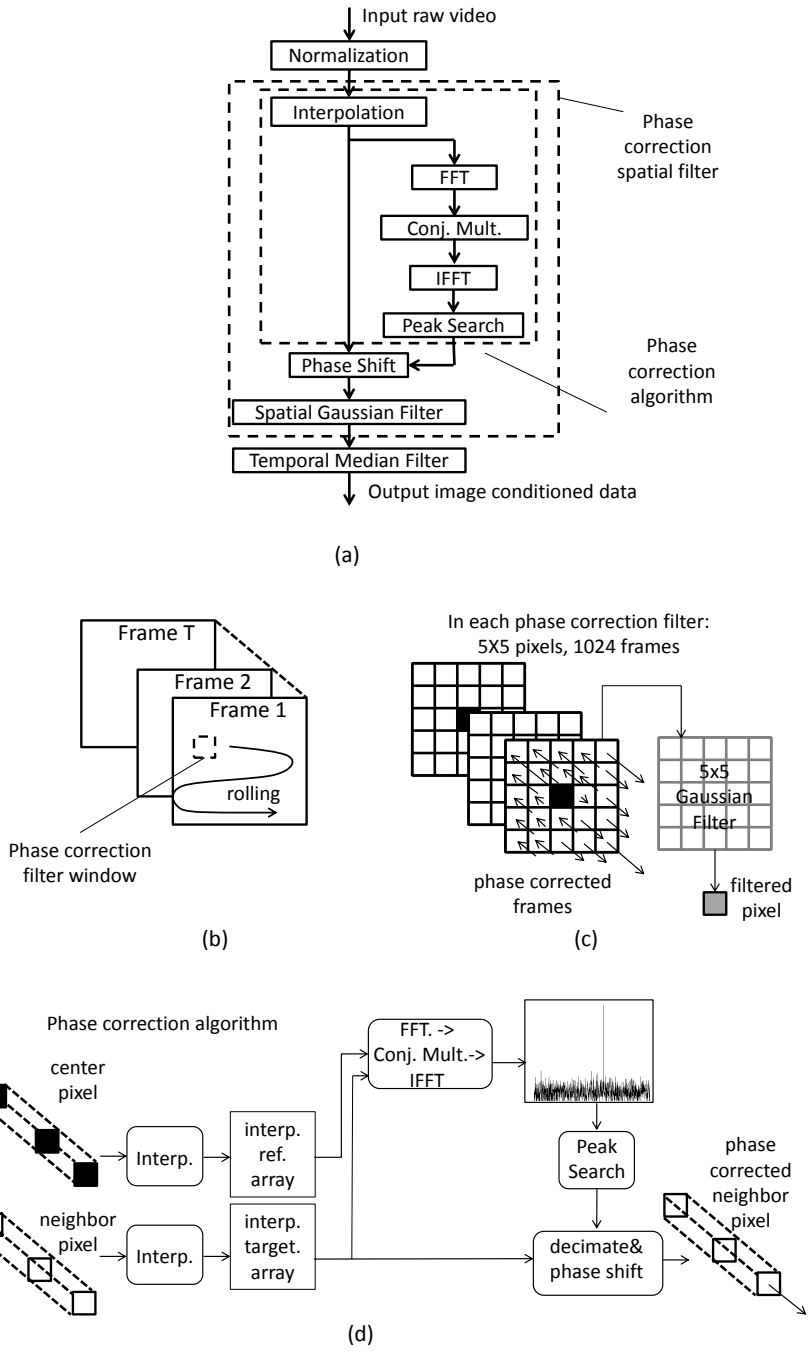


Figure 6.2. Optical mapping algorithm. (a) Overview of the image conditioning algorithm. (b) Visualization of the rolling spatial phase correction filter on the entire video data. (c) Visualization of a phase correction spatial filter window. Arrows on the pixels represent phase shifting (correction). (d) Visualization of the phase correction algorithm.

6.3.3 Phase Correction Algorithm

In order to correct the phases of the pixels, the phase difference must be computed between the center pixel and all of its surrounding neighbors in the filter window. We can calculate this difference using a bit of signal processing theory as presented by Sung *et. al* [56].

This phase correction operation is illustrated graphically in Figure 6.2 (d). First, the frame arrays are interpolated by a factor of 10 using an 81 tap FIR filter. This provides a higher resolution for phase differences. Then pairs of temporal arrays are compared, the center pixel array and a neighbor pixel array. The arrays are converted into the Fourier domain by a FFT. After that, the neighbor FFT array is conjugated and multiplied with the center FFT array. The result of the multiplication is converted back into time domain by an IFFT. The index of the pulse in the IFFT array represents the phase difference.

After finding the phase difference, the interpolated neighbor array is shifted by the relative position/time difference and down sampled by 10 to obtain the phase corrected neighbor array. Usually, the phase correction algorithm requires two long input arrays to obtain accurate phase difference result. In our implementation, the length of the input arrays is chosen to be 1024 because this is an empirically good tradeoff between the precision and runtime performance [56].

6.3.4 Temporal Median Filter

The temporal median filter is applied at the end to further remove noise after the phase correction spatial filter. The temporal median filter replaces each pixel with the median value of its temporal neighbors within a 7-element tap. After filtering, the image is conditioned and ready for analysis.

6.4 Application Partitioning

Partitioning a high throughput video application requires careful analysis at design time. Our initial design was to accelerate the software version of the algorithm developed by Sung *et. al* [56] using a FPGA. However, the algorithm operates on a second's worth of captured data at a time. This became problematic for our FPGA as the phase correction FFT would need to support a length of 32 K (1024 frames, interpolated to 10,240 frames, then padded out to 32 K frames). A single FFT core of this size would consume nearly all the resources of our FPGA. Piecewise execution of the FFT was considered, but was quickly discarded in favor of using a GPU.

Using a GPU matched well with the large array and massively parallel operations. But the frame interpolation and peak search computations are data flow barriers in the algorithm. This causes poor GPU performance. This phenomenon is discussed in [46], where a GPU implementation of the optical mapping algorithm achieves a rate about half as fast as real-time.

We chose instead to design a heterogenous system with both a GPU and FPGA. This allowed us to map the portions of the design that can benefit from deep pipelining and small buffers to the FPGA. Steps requiring large buffers with massively parallel operations leveraged the GPU. Finally, coordination, low throughput, and branching dominated tasks were assigned to the CPU. Table 6.1 shows our partitioning decisions.

The granularity of our partition is based largely on the algorithm blocks, illustrated in Figure 6.2(a). In addition to the inherent strengths of different hardware in our system, the I/O bandwidth between portions of the algorithm drove many of our design decisions. Limited bandwidth interconnects can make it challenging to quickly and efficiently transfer data between the GPU, FPGA, and CPU. Thus, we attempted to move data as little as possible while matching algorithmic blocks to the most appropriate device.

Table 6.1. Optical mapping algorithm partition decisions. Throughput definitions: $<$ 25 MB/s is low; 25 MB/s - 1 GB/s is high; and $>$ 1 GB/s is ultra high. The specific bandwidths for each computation can be found in Figure 6.3 (a).

	Interp.	Norm. Base Val.	Norm. Pixels	FFT	Conj. Mult.	IFFT	Lean Peak Search	Relative Phase Diff.	Phase Shift	Spatial Filter	Temp. Filter
Input Bandwidth	low	low	high	ultra high	ultra high	ultra high	ultra high	low	high	low	low
Output Bandwidth	high	low	high	ultra high	ultra high	ultra high	low	low	low	low	low
Accelerator Allocation	FPGA	GPU	FPGA	GPU	GPU	GPU	GPU	CPU	GPU	GPU	GPU

Video is captured using the FPGA. The FPGA also performs frame interpolation and normalization of base values. This decision was based on the fact that we can pipeline the interpolation on the FPGA so that interpolated frames would be produced concurrently with camera input.

Our FPGA-PCIe connection is limited to a single PCIe lane (bandwidth limit of 250 MB/s). Thus we represented pixels using 8 bits of precision. However, the normalization step uses 32 bit floating point numbers. To adapt, we decomposed the normalization step into a calculation of base values and normalization of pixels. We compute the the base values on the FPGA and reordered the algorithm to perform normalization on the GPU. The reordered algorithm is equivalent to the original algorithm. However, representing pixels with 8 bits introduces errors in the result. We demonstrate that the error is tolerable in Section 6.6.3.

The FFT, conjugate multiplication, and IFFT computations run on the GPU. Massive data parallelism in each *butterfly* stage of the FFT and IFFT improves core occupancy on the GPU's SIMD architecture.

Instead of calculating the relative positions between all pixels and their neighbors, we calculate partial relative positions and use the fact that they are transitive between pixel array pairs to optimize the process. It results in reducing redundant computation by $5\times$. For I/O bandwidth reasons, we perform the peak search on the GPU, but chose to allocate the relative phase difference conversion to the CPU. The phase difference conversion is a low throughput and intensively branched process aiding the peak search. We describe this optimization in Section 6.5.3.

The final processing steps are run on the GPU: phase shifting, 2D spatial Gaussian filter, and temporal median filter. The GPU already has the interpolated frame data stored in memory at this point, so it is the obvious location to shift the pixel arrays and perform filtering.

6.5 Design and Implementation

6.5.1 Overall System

The architecture of the system is shown in Figure 6.3. It illustrates which portions of the optical mapping algorithm run on which hardware. The shaded boxes encapsulate computation groups. The architecture is designed to run continuously on a system with constant camera input. Thus, it runs in a pipelined fashion. Group ① runs in a pipelined stage concurrently with groups ②, ③ and ④ in a separate pipeline stage.

Camera data is captured by the FPGA at a rate of 1024 fps and up sampled (interpolated) to 10,240 fps. Frames of interpolated data and normalization base values are DMA transferred to the host workstation's GPU over a PCIe connection. This represents computation group ①. The GPU normalizes the pixels then performs a FFT, conjugate multiplication operation, and IFFT on arrays of pixels across frames (temporally). The result of this spectral processing produces large 32 K length arrays for each pixel location. The max value in each array is found using a max peak search over all the data. The output of this group ② is the relative position of the max values in each array. These relative positions are used to calculate the absolute positioning for each pixel array. This is performed on the CPU in group ③. The CPU is used because it is faster to transfer the data out of the GPU, iterate over it on the CPU and transfer it back, than to utilize only a few cores on the GPU. Once calculated, the absolute positions are sent back to the GPU where they are used to shift each array temporally. The arrays are shifted and then down sampled back to 1024 fps. The rest of computation group ④ consists of a 2D Gaussian filter and a temporal median filter to remove noise.

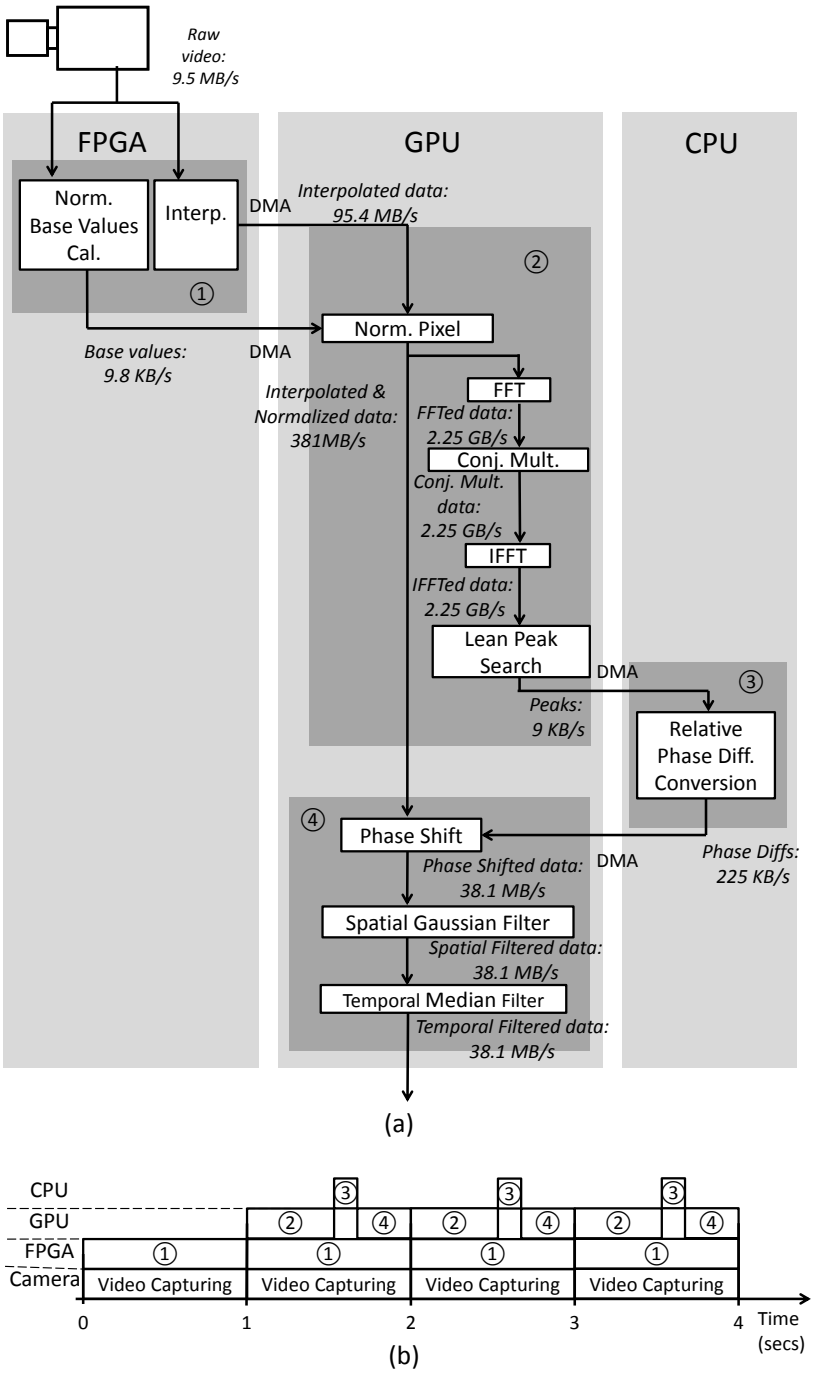


Figure 6.3. FPGA-GPU heterogenous architecture. (a) Algorithm execution diagram with throughput analysis. (b) Computation groups running concurrently in the system. Groups ①, ②, ③ and ④ are the shaded regions shown in (a).

6.5.2 FPGA Design

FPGA processing is performed in a streaming fashion. For temporal interpolation, only 8 frames of video are buffered. This buffering is necessary for the FIR filter. The most challenging aspect of the FPGA design is keeping the FIR filter pipeline full. The pixel data arrives from the camera in a row major sequence, one frame at a time. The FIR interpolation filter operates on a sequence of pixels across frames. Each interpolated frame must be produced one pixel at a time, using the pixels from the previous frames. This means filling the FIR filter with previous values for one pixel location, capturing interpolated pixels for 10 cycles, then re-filling the pipeline with a temporal sequence for another pixel location. Most of the time is spent filling and flushing the FIR filter (80 out of every 90 cycles).

To avoid this inefficiency, we parallelized the FIR filter with 9 data paths and staggered the inputs by 10 cycles. This allows the FIR filter to produce valid output every cycle from one of the 9 data paths. The output is then used to calculate the normalization base values and both are DMA transferred to the host workstation over a PCIe connection. We used the RIFFA [38] framework to connect the FPGA to the host workstation (and thus the GPU).

6.5.3 GPU Design

We designed each component on the GPU as an individual CUDA kernel. Kernels use global memory for inter-kernel coordination and for I/O data transfer. Using multiple computation dedicated kernels can improve performance over a single monolithic kernel. The data access strategies and thread dimensions can vary from kernel to kernel to more closely reflect the computation. This results in overall faster execution of all the components.

In the design of each kernel, we fully parallelized each stage to obtain the highest

GPU core occupancy. We implemented the normalized pixel calculation, conjugate multiplication, and phase shift using straight forward element-wise parallelism. The spatial Gaussian filter and temporal median filter use window/tap-wise parallelism. We used the *cuFFT* library provided by NVIDIA to implement the 32 K element FFT and IFFT operations. The peak search is implemented as a CUDA reduction, which uses memory access optimizations such as shared memory, registers, and contiguous memory assignment.

The FFT, conjugate multiplication, IFFT, and peak search are the major components of the algorithm on the GPU. Each requires ultra-high throughput and their performance is directly related to the amount of data they must process. We were able to reduce the throughput requirements for these computations, and thus improve performance, with the aid of the CPU. To do so, we created two stages, a lean peak search and a relative phase difference conversion (RPDC) to replace the original peak search stage. The lean peak search only calculates the necessary peaks by the same reduction method used in the original peak search stage. The RPDC converts the result of the lean peak search stage to the full phase difference by using the fact that relative differences are transitive. For example, we calculate the phase difference between pixel arrays a and b, and between arrays b and c using lean peak search. Let these differences be t_{ab} and t_{bc} respectively. Then $t_{ac} = t_{ab} - t_{bc}$. This optimization reduces the throughput in the FFT, conjugated multiplication, IFFT and peak search by $5\times$. The RPDC is a low-throughput computation, dominated by branching logic. This would execute with low efficiency on the GPU's SIMD architecture. We therefore implemented the relative phase conversion stage on the CPU shown as ③ in Figure 6.3.

6.6 Results and Analysis

6.6.1 Experimental Setup

We use the same experimental parameters described by Sung *et. al* [56] to guide our experiments. Input video is 100×100 resolution 8 bit grayscale video.

All our experiments are run on an Intel i7 quad-core 3.4 GHz workstation running Ubuntu 10.04. The FPGA is connected to the workstation via x1 PCIe Gen1 connector. We use a Xilinx ML506 development board with a Virtex 5 FPGA. All FPGA cores were developed using Xilinx tools, ISE and XPS, version 13.3. The GPU is an NVIDIA GTX590 with 1024 cores.

Our heterogenous design is controlled by a C++ program and compiled using GCC 4.4 and CUDA Toolkit 4.2. The C++ program interfaces with the CUDA API and the RIFFA API [38] to access the GPU and FPGA respectively. It provides simulated camera to the FPGA and coordinates transferring data to and from the FPGA and CPU/GPU.

6.6.2 Performance

Our design can execute both stages (group ① and groups ②, ③, and ④) concurrently as stage one executes on the FPGA and stage two executes on GPU/CPU. Stage one can process a second's worth of video in 0.82 seconds, at a rate of 1248 fps. However since the camera only delivers data at a rate of 1024 fps, the FPGA takes a full second to complete stage one. Transfer time is masked by pipelined DMA transfers. Thus at the end of one second, effectively all the data from stage one is in CPU memory. The GPU executes all computations in stage two in 0.86 seconds. Because an entire second's worth of data must be processed in at a time in stage two, the total latency is 1.86 seconds from the time the camera starts sending data until the time a full second's worth of processed data is available in CPU memory. This only affects latency. Both stages execute at,

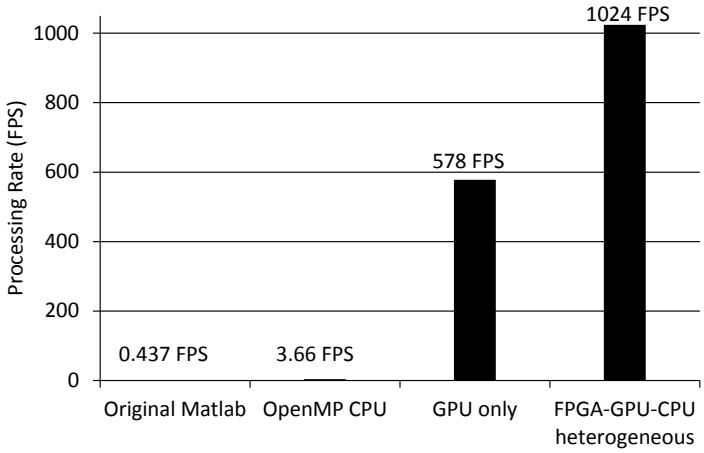


Figure 6.4. The performance of the FPGA-GPU-CPU heterogenous implementation in comparison to the original Matlab, the OpenMP C++, and the GPU only implementation.

or faster than real-time. A video of our FPGA-GPU-CPU implementation working on captured data can be found at: http://www.youtube.com/watch?v=EfvXenkiGAA.

We compare our performance against the original serial software implementation, an optimized C++ multi-threaded software implementation, and an optimized GPU implementation in Figure 6.4.

The original serial software implementation was designed and published by the Sung *et. al* [56]. The authors did not provide execution times for a full second's worth of data. However, running the same software on our i7 workstation takes 39 mins for one second's worth of data. To attempt a more fair comparison that uses all the cores of a modern workstation, we implemented an optimized C++ version (with the same algorithm implemented on the heterogeneous system). This version uses the OpenMP API to parallelize portions of the application across multiple cores. The optimized C++ program also used direct access tables to avoid computation such as trigonometric functions and the FFT output indices. This implementation took 4.6 mins to perform the same task. This is equivalent to 3.66 fps. We feel that this is an appropriate baseline for a software comparison. Our FPGA-GPU-CPU design runs 273$\times$ faster that an optimized

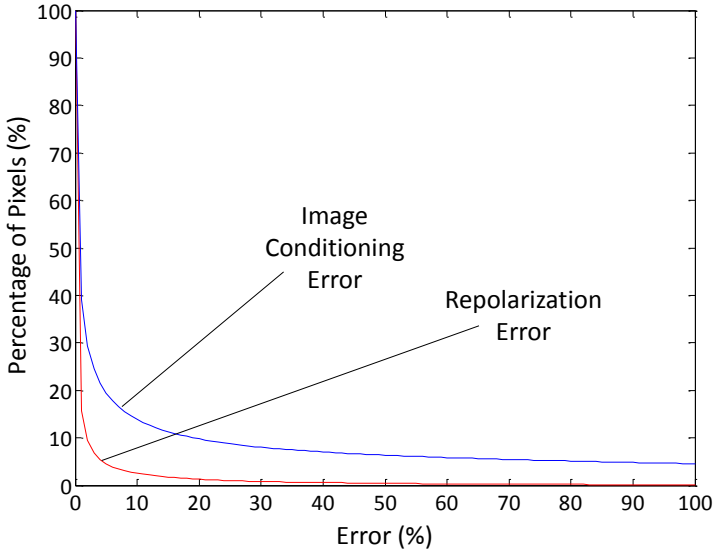


Figure 6.5. Error of the output of the optical mapping image conditioning (blue line) and error in repolarization analysis (red line). For any point (x, y) on the curve, the x represents the error in percentage scale while the y represents the percentage of pixels whose errors are greater than x.

C++ software version.

An optimized GPU implementation is described fully in [46]. It represents months of optimization tuning. It performed at a respectable rate of 578 fps. But it would have to be nearly twice as fast to achieve real-time performance. Additionally, like all the other implementations except the FPGA-GPU-CPU implementation, it would require the use of a frame capture device to be used in any real world scenario. A detail often overlooked when comparing performance.

6.6.3 Accuracy

As described in Section 6.4, the 8 bit representation of pixels (instead of 32 bit) introduces errors to the result. Algorithmic parameters limit processing of pixel arrays to those with values above 60 and with variance above 2. This limits the amount of error any one pixel can incur to 0.83 % when using 8 bits instead of 32 and rounding to the

nearest integer. However the normalization base value may be arbitrarily close to to any pixel value. Therefore, the normalized error for any pixel is unbounded. Indeed, this is evident in Figure 6.5. Some of the pixel locations show relatively significant errors. For example, about 13.8% of pixels have error greater than 10%. In practice however, we show that this is not as significant to the medical analysis.

We applied the repolarization extraction algorithm described in [56] on both the FPGA-GPU-CPU and baseline CPU implementation outputs. Figure 6.5 shows the repolarization error. This error is significantly lower than the image conditioning error. Only 2.6% of the repolarization analysis have error greater than 10%. This result indicates that using an 8 bit representation of interpolated pixels only slightly impacts biomedical features that would be extracted from the output.

6.7 Conclusion

We have addressed the challenge of real-time optical mapping for cardio-electrophysiology and presented a heterogeneous FPGA-GPU-CPU architecture for use in medical applications. Our design leverages the stream processing of a FPGA and the high bandwidth computation of a GPU to process video in real-time at 1024 fps with an end to end latency of 1.86 seconds. This represents a 273$\times$ speed up over a multi-core CPU OpenMP implementation. We also described our partitioning decisions and discussed how designs leveraging only a GPU or only a FPGA were insufficient to achieve real-time performance.

Acknowledgment

This chapter contains material printed in Field-Programmable Technology (FPT), 2012. The chapter also contains material that was omitted from the publication due to space constraints. The work described in this chapter is a collaboration with Pingfan

Meng. The dissertation author was not the primary investigator. The dissertation author was the second author of this paper.

Chapter 7

Hardware Accelerated Online Boosting for Multi-Target Tracking

7.1 Introduction

Robust object tracking is a critical component for many applications. Input and gesture recognition for human-device interaction [9], autonomous vehicle systems [6], and video surveillance [54] all require accurate tracking input. Many of these practical applications require tracking multiple independent targets at once, with low latency, several times a second. Tracking effectively translates high bandwidth sensor information into a low bandwidth set of data points for higher level algorithms. This is a challenging task because an object's appearance can change over time. Small changes in lighting, occlusions, deformations, or rotations can have a dramatic effect.

Boosting, as an approach, has been employed in machine learning applications with considerable success. In the computer vision community, classifiers of boosted Haar features are commonly used as face and object detectors [58]. Training these classifiers is typically performed offline with many training examples over several rounds. This approach is based on the idea that a large volume of training examples be collected and used for training, a priori.

Research has shown that online boosting can be very effective for object tracking

[30, 7]. In contrast to traditional offline boosting, online boosting gathers examples at runtime and trains a classifier incrementally. This approach provides training examples from the current environment and can result in a more adaptive and accurate classifier. Offline boosting models rely on a large number of examples to effectively cover the space of possible representations. But online boosting approaches only need examples of the current representation to build an effective model for the object being tracked. This can reduce the number of examples needed for training and improve classifier accuracy at the same time.

Online training allows the appearance model to adapt as the object changes. This however, incurs additional computation. Performance requirements can be difficult to meet when only evaluating a fixed classifier. Online boosted tracking algorithms must evaluate and train the classifier under the same performance requirements.

In this paper, we consider the task of accelerating an online boosting based tracker capable of tracking multiple independent targets. We evaluate the tracking algorithm proposed by Babenko [7]. We propose and implement two hardware designs using a GPU and a FPGA. These designs are compared against software-only implementations. The main contributions are:

- A FPGA design for training and evaluating an online boosted tracking classifier.
- A GPU design for training and evaluating an online boosted tracking classifier.
- A comparison and analysis of the hardware accelerated designs.

We continue with a discussion of related work. Afterwards, an explanation of the online boosting algorithm is provided. We discuss our tracking application and its requirements. This is followed by descriptions of the hardware accelerated designs. Experimental performance results and analysis follow. We close with conclusions.

7.2 Related Work

Much of the research in online boosting for tracking focuses on improving the algorithms [30, 59, 7]. While there are numerous hardware accelerated tracking applications, they focus on the evaluation of trained classifiers, not the training. Online boosting requires evaluation and training to be completed at runtime. The complexity and iterative dependency of training makes it difficult to parallelize. To our knowledge, there are no hardware accelerated designs for online boosted tracking in the literature at the time of this writing.

Accelerating boosted classifier training has been addressed by Lo [45]. They present a FPGA based architecture to reduce the time to train Viola-Jones style classifiers. They achieve a $14\times$ speed up over a high end processor. This work is similar to our own, but deals only with accelerating offline training.

Heinzle et al. describe their work to accelerate computational stereo camera processing in [34]. Their design uses a FPGA, GPU and CPU to process the stereo data in real time. Online boosted tracking is employed in their framework. However the tracking is run on the CPU. The authors point out that their system would benefit from a FPGA accelerated online boosting tracking implementation.

Coates et al. propose a GPU based accelerated system for robotic object detection [20]. Their system uses a boosted classifier and stereo disparity maps for robotic sensing. Evaluation of their classifier is performed on the GPU, but training is performed on a cluster of CPUs.

7.3 Algorithm

The tracking algorithm we employ is proposed by Babenko [7]. It is an online boosting algorithm for tracking based on MILBoost [59]. It was selected because of its

Algorithm 2. Tracking Algorithm

Input: New image frame at time t

1: Select a set of samples, cropped from frame
$X^s = \{x | s > \|l(x) - l^*_{t-1}\|\}$
2: Calculate Haar feature values, $f(x)$, for $x \in X^s$
3: Use classifier, $H(x) = \sum_k h_k(x)$, to classify samples X^s
4: Set new location $l^*_t = l(\text{argmax}_{x \in X^s} H(x))$
5: Select positive and negative samples sets from frame
$X_1 = \{x | r > \|l(x) - l^*_{t-1}\|\}$
$X_0 = \{x | q \leq \|l(x) - l^*_{t-1}\| < q\prime\}$
6: Train classifier on positive and negative sets
$H = Train(X_1, X_0)$

robust appearance model. The algorithm consists of two steps: 1) find and update the new target location, then 2) update the trained classifier.

For each new image frame, a region surrounding the last known location is evaluated. Evaluation yields a new location. Then the region surrounding the new location is used to to select positive and negative training examples. These examples are used update the classifier which will be used for the next frame. This tracking flow is illustrated in Algorithm 2.

The algorithm is a *two pass* algorithm in the sense that it must access the image frame twice. First to search for the new location. Then again, after the new location is found to gather training examples. The passes must take place sequentially. The second pass cannot begin until the new location has been found. A sequential dependency exists between image frames as well. The next frame cannot be evaluated until the classifier has been trained from examples drawn from the current frame.

We continue with a detailed explanation by describing the three basic components common to most tracking algorithms: the motion model, the search strategy, and the appearance model.

7.3.1 Motion Model

The motion model assumes the object location at time t will be within a radius, s, from the previous object location at time $t-1$. It grants equal probability to each possible location within radius s. If we denote each possible location as $l(x)$ and the object location l^*, then:

$$p(l_{t_1}^*|l_{t_0}^*) \propto \begin{cases} 1, & \text{if } \|l_{t_1}^* - l_{t_0}^*\| < s. \\ 0, & \text{otherwise.} \end{cases} \tag{7.1}$$

7.3.2 Search Strategy

As all possible locations within the search radius, s, are equally likely, Equation 7.1 implies a uniform search strategy. In our implementation, all the possible locations within the radius are evaluated. The location, $l(x)$, with the maximum classification value is selected as the new object location, l^*. This greedy strategy lends itself well to parallel execution as there is no data dependency between locations during evaluation. More sophisticated algorithms such as particle filters have been used for online boosting [60]. However particle filtering would introduce additional dependencies between data locations.

7.3.3 Appearance Model

The basic idea of boosting is to combine several weak classifiers, h, into a strong classifier, H. This is achieved by iteratively maximizing the log likelihood of the strong classifier:

$$\log \mathscr{L}(H_{k-1} + h). \tag{7.2}$$

At each iteration, k, the existing strong classifier is combined with the next weak

Figure 7.1. A. Circular search region with radius s (left). **B.** Circular region with radius r for positive examples and sampled annular region with radii q and $q\prime$ for negative examples (right).

Algorithm 3. Appearance Model Training Algorithm

Input: Labeled positive and negative sets $\{X_1, 1\}, \{X_0, 0\}$ where $X_i = \{x_{i,1}, x_{i,2}, ...\}$

Calculate Haar feature values, $f(x)$, for $x \in X_1, X_0$
Update parameters μ_1, σ_1, μ_0, σ_0 for all features
Calculate weak classifier predictions, $h_m(x)$, for all M weak classifiers, for all samples X_1, X_0
Initialize $H_{1,j} = 0$, $H_{0,j} = 0$ for all j in X_1, X_0
Select K best weak classifiers from pool of size M
for k = 1 to K **do**
    for m = 1 to M **do**
        $p^m_{i,j} = \sigma(H_{i,j} + h_m(x_{i,j}))$
        $p^m_i = 1 - \prod_j (1 - p^m_{i,j}))$
        $\mathscr{L}^m = \frac{1}{|X_1|}\log(1 - p^m_1) + \frac{1}{|X_0|}\log(p^m_0)$
    end for
    $m^* = \text{argmax}_m \mathscr{L}^m$
    $\mathbf{h}_k \leftarrow h^*_m(x)$
    $H_{i,j} = H_{i,j} + \mathbf{h}^*_k(x)$
end for

Output: $H(x) = \sum_k \mathbf{h}_k(x)$

classifier, h, that maximizes this quantity over the training data. Training data comes in the form of examples, x_i, with binary labels, $y_i \in \{0, 1\}$.

Instance (location) probability is measured as:

$$p(y_i = 1|x_i) = \sigma(H(x)) \tag{7.3}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$, the sigmoid function. This is the probability that location x_i is a positive sample (represents the object).

The algorithm makes use of an additional technique called Multiple Instance Learning [22] to improve classifier robustness. The idea is to group samples together into sets (called *bags*) and train using the sets instead of the samples directly. The training data has the form $\{(X_i, y_1), ..., (X_n, y_n)\}$, where $X_i = \{x_{i,1}, ..., x_{i,m}\}$. Set labels are negative unless at least one sample in the set is a positive example. Then the set label is positive. Each sample inherits its set label. In practice we use two sets: a positive set and a negative set.

Using Multiple Instance Learning has two main benefits. It mitigates the problem of correctly labeling samples as they are collected. It also provides a higher degree of flexibility in finding the decision boundary during training.

The first benefit is achieved by defining the positive training set as all the samples in a radius, r, around the newly found object location. This radius must be small enough to avoid including invalid positive samples in the positive set. It must also be large enough to provide a margin of error for the newly found object location. We define the negative training set by selecting samples from an annular region between two radii, q and $q\prime$, surrounding the newly found location. The outer radius $q\prime$ is larger than the inner radius q. The inner radius is set appreciably larger than r to reduce the chance of including any positive samples. See Figure 7.1 for an illustration of these regions.

The second benefit deals with the inherent ambiguity in selecting a positive example from image data. Even when performed by a human, the best single positive sample can be slightly incorrect (e.g. cropping an image by hand). Providing multiple overlapping samples with positive labels can be a better solution. However, training directly on these samples can confuse the classifier and reduce discriminative power. To preserve accuracy and flexibility in finding the decision boundary, training is better served when performed on the bags instead.

To train on the bags, the algorithm needs to be able to represent bag probability.

This is accomplished using the Noisy-OR model. It defines bag probability as:

$$p(y_i = 1|X_i) = 1 - \prod_j (1 - p(y_i = 1|x_{i,j})). \tag{7.4}$$

This model has the property that if one of the samples has high probability, then the entire bag will have high probability. When calculating likelihoods, the veracity of each instance label is not important. Only the bag label matters. Combined with Equation 7.3, the classifier can be trained using the log likelihood of bags instead of instances. Algorithm 3 lists the steps for the training algorithm.

The algorithm uses Haar-like features, similar to those used for face and object detection [58]. A Haar feature is a collection of weighted rectangular regions defined within a window. The rectangles define a region of the window over which to sum pixel values. The summing is typically performed efficiently using an integral image. The value of the feature, $f(x)$ is the weighted sum of its rectangular regions. Haar-like features differ from Haar features in that they have between two and six rectangles each, and rectangles need not be adjacent. In the rest of the paper, we refer to these features simply as Haar features.

At initialization, a pool of M Haar features are generated with random rectangle coordinates and weights. During training, K of these features are selected and are used as weak classifiers, $h(x)$, to form the strong classifier, $H(x) = \sum_k \mathbf{h}_k(x)$. Each weak classifier consists of a Haar feature and four additional parameters: μ_1, σ_1, μ_0, σ_0. These additional parameters quantify the degree to which the feature represents the object being tracked. They are updated during training as

$$\mu_i = \gamma\mu_i + (1 - \gamma)\frac{1}{n}\sum_{i|y_i=i} f(x_{i,j}) \tag{7.5}$$

$$\sigma_i = \gamma\mu_i + (1-\gamma)\frac{1}{n}\sum_{i|y_i=i}(f(x_{i,j}) - \mu_i)^2 \tag{7.6}$$

using positive and negative samples, $x_{i,j}$.

At runtime, each weak classifier predicts the likelihood of a sample window, x using the log odds ratio:

$$\begin{aligned} h(x) = &\frac{1}{2\sigma_1}(f(x)-\mu_1)^2 + \log(\frac{1}{\sqrt{\sigma}_1}) \\ &-\frac{1}{2\sigma_0}(f(x)-\mu_0)^2 - \log(\frac{1}{\sqrt{\sigma}_0}). \end{aligned} \tag{7.7}$$

This formula results in higher scores the closer a sample's feature value is to the mean of the positive examples.

7.4 Tracking Application

We demonstrate the hardware accelerated online boosting tracker via a human computer interaction (HCI) application. The requirements are to track at least three independent points at a rate of at least 30 frames per second (FPS). We selected 30 FPS because it is the sampling rate of most consumer video cameras. We need at least three independent tracking points so that we can track a human and two hands. Supporting additional points will improve fidelity and robustness by tracking finger tips and other body parts. In future applications, for example in autonomous vehicle settings, supporting multiple points will be crucial.

The implementation must also evaluate each image frame at three different scales. The current scale and one scale higher and lower than the current scale. This is to be able to track objects as they move closer and further from the camera.

The algorithm was initially implemented on a PC using software. It was unable

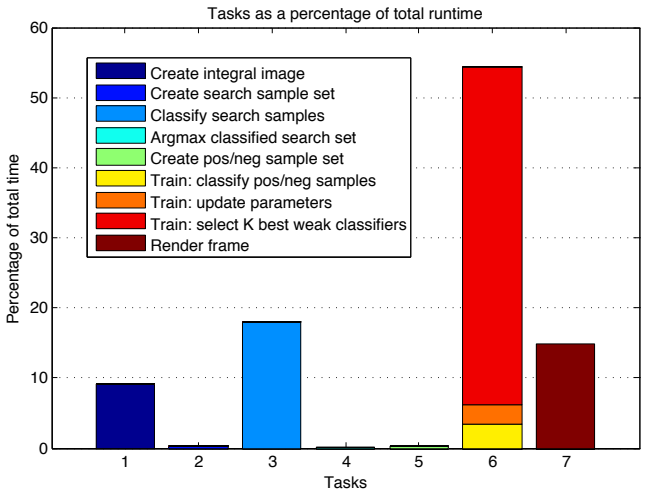


Figure 7.2. Performance of different tasks in the algorithm as a percentage of total time.

to meet the project goals. Profiling revealed the bottlenecks to be: creating the integral image, evaluating samples in the search sample set, training the classifier, and rendering the image. See Figure 7.2. To achieve the project goals and reduce the bottlenecks, we evaluate the algorithm in parallel hardware.

7.5 Hardware Design

The hardware designs largely address the bottlenecks identified in Figure 7.2. We focused on accelerating tasks labeled 1, 3, and 6. However, because rendering images is not strictly required for the application, we did not explore acceleration for that task.

The largest amount of time (54.5%) is spent in training the classifier; specifically in the nested loops starting on line 5 in Algorithm 3. This task is predominately sequential. Each of the K outer loop iterations must wait for the previous iteration to complete so that $H_{i,j}$ is valid before starting the next iteration. The calculation of the log likelihoods on lines 7 and 8 can be parallelized, but line 9 acts as a barrier which limits pipelining opportunities. This represents the inherent data dependency in boosting applications. It also represents the largest acceleration opportunity in our hardware designs.

7.5.1 FPGA Design

The FPGA design is partitioned between PC software and FPGA hardware. The software is responsible for acquiring image frames, passing data to the FPGA, receiving data from the FPGA, and keeping track of tracking state. PC-FPGA communication is achieve using the RIFFA framework [39].

The algorithm initializes in software. A pool of Haar features is generated and tracking parameters are initialized by selecting a subset for the initial classifier H. The software also handles target registration. Tracking locations can be of any size. Input images are scaled to fit a fixed hardware window size of 20×20.

For each new 8 bit grayscale image frame, a square region of the frame surrounding the current location is cropped and sent to the FPGA along with parameters. The square region includes only the pixels falling within the sample radius s. The parameters define the Haar features, weak classifiers, search radius, scaling factors, and inform the FPGA which operations to perform. The FPGA design is illustrated in Figures 7.3a and 7.3b. The algorithmic flow and component details follow.

Update Location

Parameters and image data sent from the PC are received by the *Main* module on the FPGA. The Main module coordinates running the three stage pipeline as pictured in Figure 7.3a. In this invocation, only *Stage 0* is run. Stage 0 streams frame data through three parallel classification pipelines. Each classification pipeline scales the frame data independently, converts the scaled data into integral image data, and runs it through a systolic sliding window.

Classification Pipeline Image scaling is used to resize the input region so that the tracked location fits within a 20×20 window. It is also used to enable evaluation at one

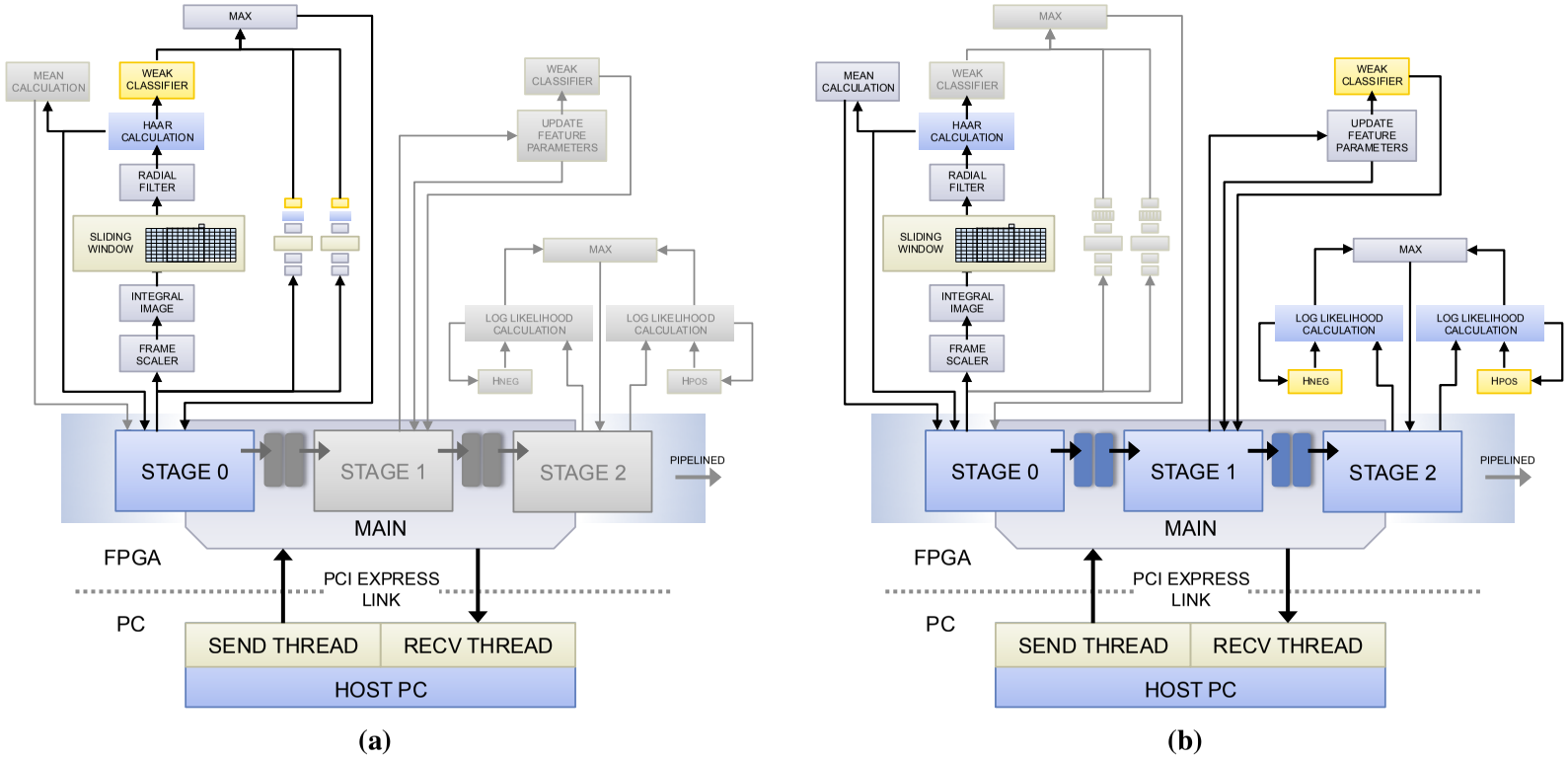


Figure 7.3. High level architecture of FPGA design. Figure (a) shows the modules active during location update. Grayed out modules from Stages 1 and 2 are inactive during location update. Figure (b) shows the active modules during classifier training. Only one classification pipeline in Stage 0 is active during classifier training.

scale above and below the current scale. Because the same input data stream is sent to all three pipelines, the data must be large enough to accommodate the largest scaling factor. This is calculated on the PC and the cropped region is set accordingly.

The scaling module uses block RAM (BRAM) to buffer lines of the cropped image data. Four pixels are used to interpolate each output pixels at the correct scale. An integral image is calculated on the scaled image. Only the 17 least significant bits of each integral image pixel are kept. This has no affect on Haar feature calculation as higher order bits will be subtracted away.

Integral image pixels are streamed to a sliding window architecture, similar to that employed by Cho [19]. It is a systolic architecture where each new pixel generates a new vertical column of pixels in the window. BRAMs buffer horizontal lines until a full window has been buffered. A register array is used to maintain the current window values so that any arrangement of pixels can be accessed each cycle.

The register window is accessed by six parallel Haar feature extractor modules. This allows each classification pipeline to calculate one Haar feature every cycle. Haar calculation involves summation and multiplication for the rectangle weights. In software, the weight parameter is represented using a single precision float. To avoid the floating point expense in hardware, we represent rectangle weights using 4 bits of precision. This decision was arrived at empirically after evaluating several fixed bit width representations for rectangle weight. See Figure 7.4. This yields Haar values at the same level of accuracy as when using a single precision float.

Radial and annular filtering is accomplished by tracking window positions within the cropped region. Windows meeting the radial criteria have all K Haar values calculated in K cycles.

After the Haar features are calculated, they are streamed to the *WeakClassifier* module which performs the calculations in Equation 7.7. The WeakClassifier calculates

values in a fully pipelined fashion. This produces a new value every cycle. The values are summed to achieve a single value per window location. The maximum sum across all three classification pipelines is returned to the PC along with the corresponding window location and scale. The PC uses this data to update the target location.

We avoid division, square root, and logarithm operations in the WeakClassifier module by precomputing these functions on the PC. This leaves only multiplication, addition, and subtraction. While this can be performed efficiently using fixed point arithmetic, we use floating point operator modules. This is done because the weak classifier feature values take on a wide range of values. The algorithm's ability to discriminate between these values directly impacts its accuracy. Using fixed point representation would require considerably more bits of precision.

Classifier Training

Training the classifier requires most of the same data needed for the update location operation. However, this data path has data barriers imposed by the algorithm. Training depends on the selected features from the previous iteration and thus cannot overlap. Log likelihood values calculated in each training iteration require all the weak classifier predictions to have been calculated. These weak classifier predictions cannot be calculated until all the feature parameters have been updated. Finally, the feature parameters cannot be updated until all the Haar values have been calculated. To accommodate these barriers, we split the data path into three *Stages* that can run concurrently without data dependencies. Data generated between Stages are stored in FIFOs. This data is generated in both sample major and feature major order, depending on the Stage. Data sent to the intermediate FIFOs is stored according to how it will be accessed in subsequent Stages.

As before, Stage 0 calculates Haar feature values for each sample location. These

are the Haar values represented in line 1 of Algorithm 3. In addition, the mean $E(X)$ and squared mean, $E(X^2)$ for each Haar feature are calculated incrementally as the values are generated. Division is accomplished using floating point operators.

Stage 1 uses the saved Haar values, $E(X)$, and $E(X^2)$ to update feature parameters μ_1, σ_1, μ_0, σ_0 and calculate weak classifier prediction values for each sample location. The feature parameters are updated according to Equations 7.5 and 7.6 and sent to the PC. In doing so, $E(X)$ is used to update μ in a straightforward manner. Unfortunately, the variance term in Equation 7.6 is calculated with respect to the *just updated* μ value, not $E(X)$. Typically one would use the following definition to calculate this variance term, $var(X) = E(X^2) - (E(X))^2$. But the variance term is actually a *relative variance* value and $\mu \neq E(X)$. Fortunately, defining variance in terms of expected value provides a solution that avoids (re)iterating over the Haar values. We calculate the relative variance term in Equation 7.6 as:

$$E(X^2) - 2\mu E(X) + \mu^2. \tag{7.8}$$

A proof of this relative variance representation follows:

Let X be a discrete random variable and μ be any real value. The variance of X relative to μ can be expressed as:

$$relvar(X, \mu) = E(X^2) - 2\mu E(X) + \mu^2$$

Proof:

$$
\begin{aligned}
relvar(X,\mu) &= \sum_{x\in X}(x-\mu)^2 p(X=x) \\
&= \sum_{x}(x^2-2\mu x+\mu^2)p(X=x) \\
&= \sum_{x}(x^2-2\mu x)p(X=x)+\mu^2\sum_{x}p(X=x) \\
&= \sum_{x}(x^2-2\mu x)p(X=x)+\mu^2 \\
&= \sum_{x}x^2p(X=x)-2\mu\sum_{x}xp(X=x)+\mu^2 \\
&= E(X^2)-2\mu E(X)+\mu^2
\end{aligned}
$$

After Stage 1 generates all the weak classifier prediction values, *Stage 2* uses these values to calculate the log likelihood for each feature. The log likelihood is calculated using a pipeline of floating point operators. This preserves accuracy for the wide range of resulting values with operations: exponential, division, and logarithm. Stage 2 iterates K times, selecting the feature with the maximum log likelihood each time. The intermediate classifier $H_{i,j}$ is stored and updated each iteration.

Because of the data dependency between iterations, each iteration's processing cannot be parallelized. However, the log likelihood calculation is not data dependent. In our design we parallelize this data path 8 times. Positive and negative samples are calculated in their own set of parallel pipelines. Thus there are 16 parallel pipelines. This factor was selected to balance the runtime between Stages. Stage 2 generates K numbers in total, representing the selected top K weak classifiers h_k in H. The selected features are sent to the PC to update the definition of the trained classifier for the next frame.

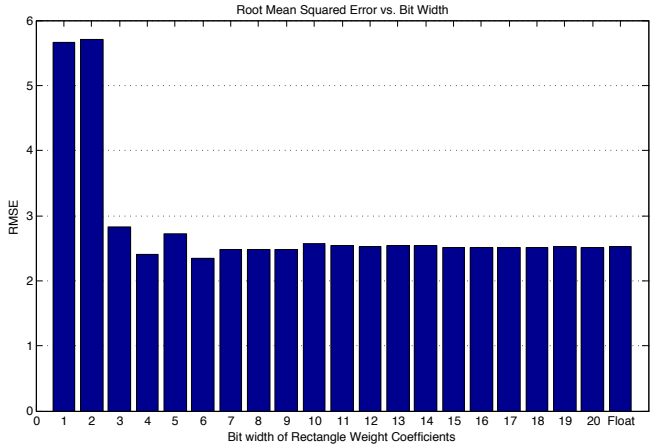


Figure 7.4. Root mean squared error of different bit widths for Haar rectangle weights over test sequences.

7.5.2 GPU Design

The GPU design consists of two CUDA kernels, invoked from a C++ application. The functions in these kernels were selected based on how amenable they are to GPU execution. These kernels perform feature calculation and log likelihood calculation respectively. The rest of the algorithm runs in software.

For each new 8 bit grayscale image frame, two additional scaled frames are generated. Integral image representations of all three frames are calculated. Sample locations surrounding the current location are then passed to the *feature calculation* kernel along with the integral images, Haar, and feature parameters. The kernel performs Haar feature extraction and weak classifier prediction on all specified samples as in lines 2-3 in Algorithm 2. It returns the location with the maximum prediction value.

Using the new location, the CPU samples two sets of positive and negative training locations. It passes these locations to the feature calculation kernel. However, during this invocation the kernel only calculates and returns Haar values for the specified samples. These values are used by the CPU to update feature parameters as in Equations 7.5 and 7.6. Once updated, these parameters are sent to the feature calculation kernel. This

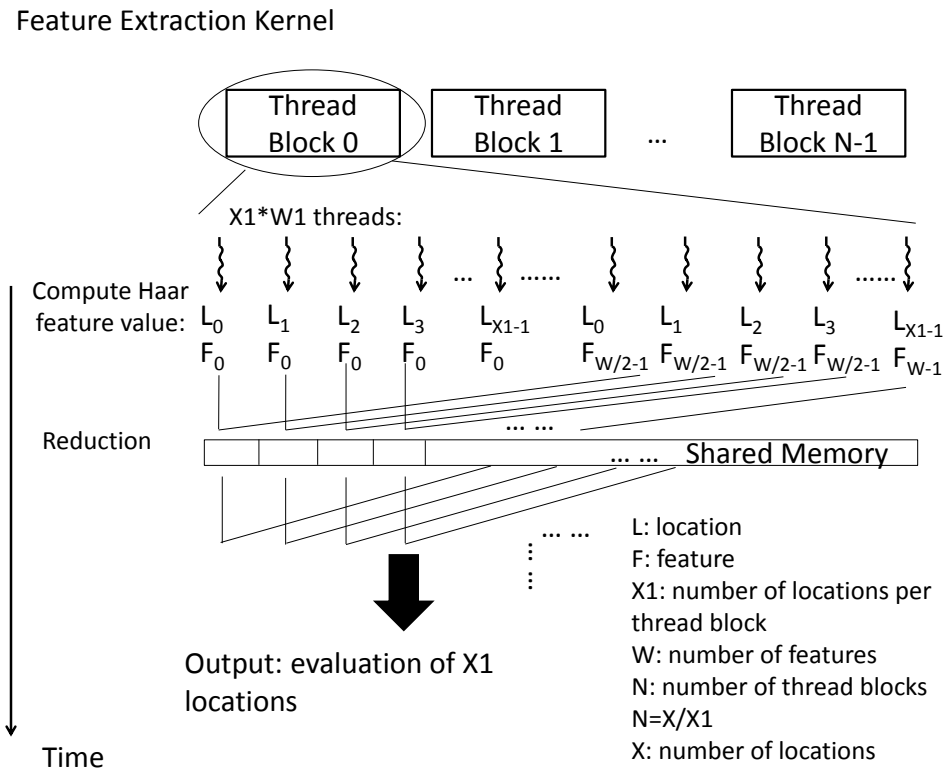


Figure 7.5. Implementation of the feature extraction kernel.

invocation of the kernel is also different in that it only calculates weak classifier prediction values. It returns nothing to the CPU, but keeps these values in GPU memory. These last two invocations of the feature calculation kernel are in some sense executing the first half and second half of the kernel respectively.

The prediction values are used by the *log likelihood calculation* kernel. This kernel performs the calculations on lines 7-9 in Algorithm 3. It takes as input, a partially boosted classifier on each invocation and returns log likelihood values for each feature. Software sorts these values to find the next weak classifier $\mathbf{h}_k^*$. It updates the intermediate classifier $H_{i,j}$ and continues iterating. Figure 7.6(b) illustrates this arrangement.

Feature Calculation

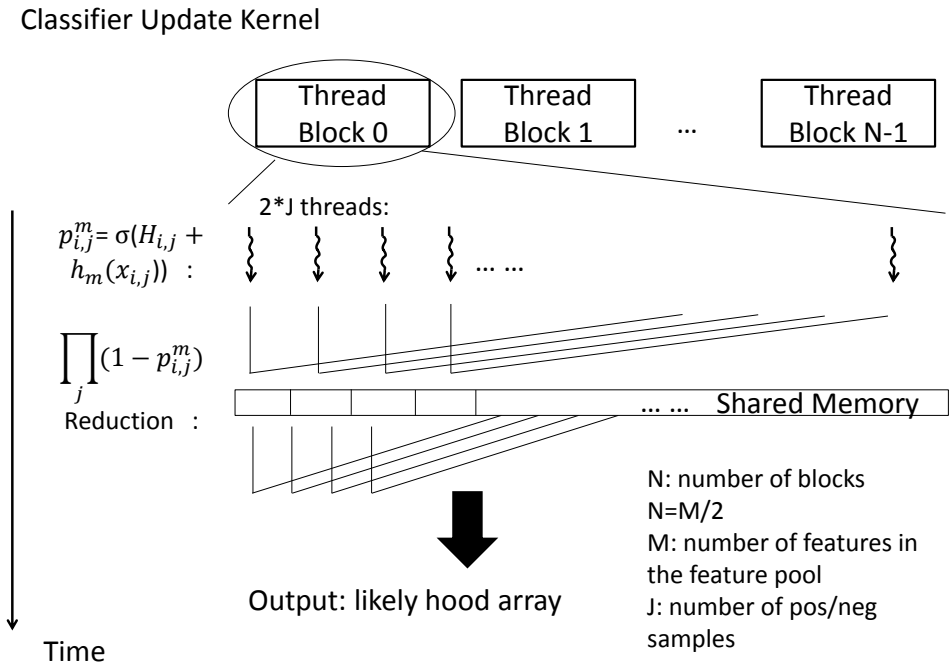
The feature calculation kernel is designed to exploit the lack of data dependency between sample locations. For X samples and W features, it utilizes $X \times W$ CUDA threads to compute values in parallel. Each CUDA thread computes a single feature value for one location. To generate the prediction value for a sample, its feature values must be summed. This is accomplished using a reduction.

The number of features limits the number of threads accessed by each reduction in a thread block. In our design $W = 50$. As this is well below the maximum number of threads in a thread block, the reduction only uses shared memory. No global memory synchronization is needed and thread blocks can run independently. In order to maintain high GPU streaming multiprocessor occupancy, the kernel processes multiple locations on each thread block. Threads are aligned along the location index to achieve a sequential addressing reduction. This keeps processing threads near each other and within the same warp. Figure 7.5 illustrates the CUDA thread arrangement. After the all the feature values are calculated, another reduction selects the maximum value across all samples.

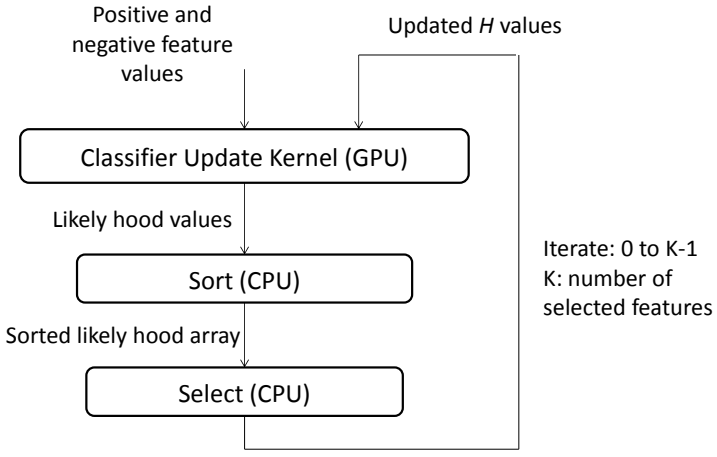
Log Likelihood Calculation

The log likelihood kernel calculates log likelihood values for a single iteration of the training loop as listed on line 5 of Algorithm 3. The kernel is designed to only calculate values for a single iteration because of the dependency between iterations. Iterating within the CUDA kernel would require a global synchronization of all threads to implement the sorting operation. This is inefficient for the GPU and comparatively slow. Therefore, the CPU sorts and selects features from the GPU kernel output between invocations.

Within a single invocation, the kernel utilizes $M \times J$ parallel threads to calculate the instance probabilities for M features across J training samples. This computation



(a)



(b)

Figure 7.6. Implementation of the feature update kernel. (a) Thread assignment; (b) Sequential iteration data flow.

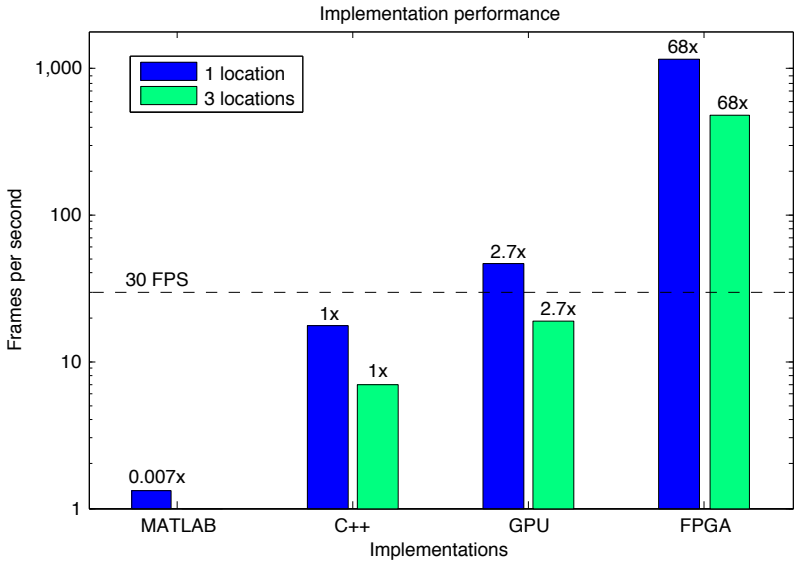


Figure 7.7. Performance of implementations. Speed up factors over the C++ implementation are shown above each bar. Y axis is logarithmic.

is equivalent to line 7 of Algorithm 3. A reduction is used to compute the positive and negative likelihoods as listed in line 8. This reduction is similar to the reduction in the feature calculation kernel in that intermediate data is stored using shared memory. The CUDA thread assignment is illustrated in Figure 7.6(a).

7.6 Results And Analysis

We tested all our implementations on the same PC, a 4 core Intel i7 3.6 GHz system with 16 GB RAM, running Ubuntu 10.04. Except for the MATLAB implementation, we tested the implementations using C++ applications with O2 GCC optimizations. 640x480 resolution image frames were used. We used the OpenCV library for image manipulation functions. For all experiments, the feature pool size was set to 250, with a classifier size of 50. We set s to 35, r to 4, q to 6, and $q\prime$ to 8. Figure 7.8 shows some tracking sequences from our experiments.

Figure 7.7 shows the performance of our implementations. Because many practi-

cal applications do not need to render the image frame, we did not include the time to render in our measurements. Otherwise, time measurements for the performance reported include the time for running the entire algorithm, not just the accelerated kernels. We discuss kernel performance relative to overall performance in the sections below.

7.6.1 Software-only

We implemented the algorithm in software using Mathworks MATLAB and C++. The MATLAB implementation is a single threaded implementation that uses MATLAB APIs exclusively. It is capable of running the algorithm at 0.123 FPS. We did not attempt to run the algorithm with multiple targets.

Our C++ implementation is a highly optimized, multi-threaded software implementation that makes use of Intel Integrated Performance Primitives vector instructions. It also uses the OpenMP library to parallelize the application. This implementation is based on a version provided by the authors of the algorithm [7]. It is capable of running at 17 FPS while tracking a single target and at 7 FPS when tracking three targets concurrently.

7.6.2 GPU

Our GPU implementation was written using the NVIDIA CUDA 5.0 SDK and was run using a NVIDIA GeForce GTX 690. It runs at a base frequency of 910 MHz and has 3072 CUDA cores. This design runs the algorithm partially in C++ and partially on the GPU. The implementation is capable of running the tracking algorithm on a single target at 47 FPS. When tracking three independent targets, the frame rate is 19 FPS. When tested separately, the feature calculation kernel runs at 92 FPS. Similarly, the log likelihood calculation kernel runs at 137 FPS (rate includes all 50 invocations) .

Table 7.1. FPGA design resource and VC707 utilization.

Slice Reg.	Slice LUT	BRAM	DSP48E
187505	231971	224	304
31%	76%	22%	11%

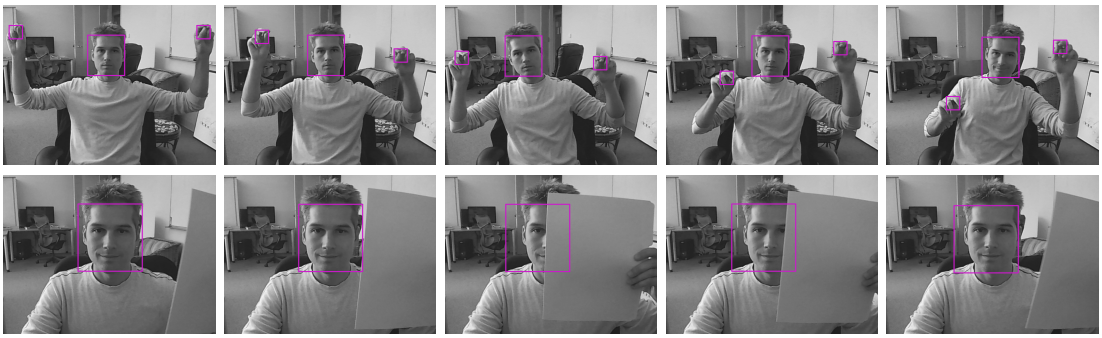


Figure 7.8. Example tracking sequences (left to right). Multiple target tracking.

7.6.3 FPGA

The FPGA design was built using Xilinx Vivado 2013.3 in Verilog. High level synthesis was not used. It was implemented on a Xilinx Virtex 7 VC707 development board and run at 250 MHz. It has a x8 Gen 2 PCIe connection to the PC. Table 7.1 lists the resource utilization of the entire design. Values between the FPGA implementation and the software implementation differ by a maximum of 0.77% (on the log likelihood calculation). This is due to the differences in floating point operators between the CPU and FPGA and has no affect on the selection of features.

The FPGA implementation communicates with a C++ application. However, as all the image processing is performed on the FPGA, the software only performs bookkeeping functions. It is capable of tracking a single target at 1160 FPS, three targets at 480 FPS, and 57 targets at 30 FPS. When tested separately, the invocations to find the new location and to train the classifier run at 6635 FPS and 1383 FPS respectively.

7.6.4 Comparison

The GPU implementation runs 2.7× faster than the C++ implementation, whereas the FPGA implementation does so 68× faster. The difference in performance between the two stems from differences in their respective architectures. The FPGA design is a pipelined hardware data path with dedicated operators to calculate values on on manually scheduled data. This results in a substantial advantage over the massively parallel automatically scheduled GPU execution.

In a GPU, each thread is configured to run a specific kernel on potentially multiple data. This is efficient when there exists enough data to process. However as the data becomes exhausted, threads sit idle waiting for the rest to complete. This is generally unavoidable in applications. But the duration and amount of idle threads can have a significant impact on performance. Data dependencies, barriers, branches, and reductions can all contribute to idle threads. Both our kernels suffer from this behavior. The feature extraction kernel suffers the most due to the Haar and prediction value sum reduction. Figure 7.9 illustrates this graphically. During the Haar value sum, the number of active threads decreases by half after each round of threads complete. Afterwards, the prediction

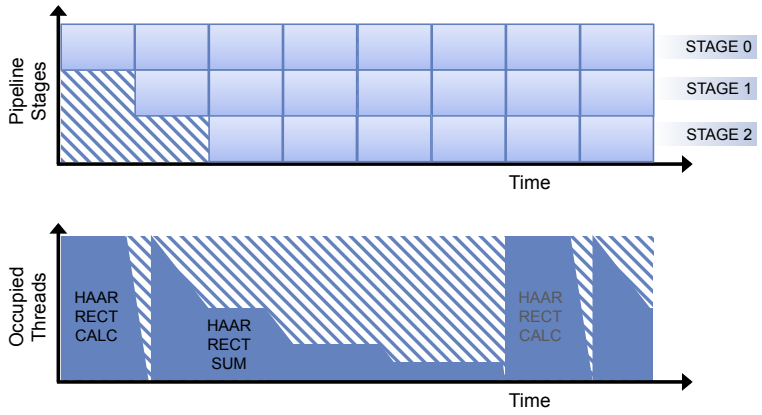


Figure 7.9. FPGA pipeline filling (top). GPU non-idle threads for feature calculation kernel (bottom). Hatched region represents idle resources.

value sum underutilizes the GPU by only occupying a fraction of the available threads. This underutilization cannot be solved by including more data from additional targets because the design must accommodate a variable number of tracking targets. Thus each thread must run to completion before acquiring new data from the next target. A design that expected a specific number of tracking targets could mitigate this effect, but not eliminate it.

On the FPGA, multiple targets are processed serially as with the GPU. However, Stage modules are able to overlap in time. This contributes most significantly to the better performance. The amount of idle time is limited to the fill and drain time for the three Stage pipeline. Since each Stage's runtime is on the order of microseconds, this idle time is minimal. Additionally, the data path has been manually parallelized at bottlenecks to keep each Stage's runtime nearly the same. Lastly, the algorithm executes almost completely on the FPGA. The CPU is only responsible for bookkeeping operations. The GPU design requires more interactions with the CPU and relies on the CPU for some image processing. This manifests in lower overall performance for the GPU design despite respectably high individual kernel performance.

Unlike the GPU implementation, the performance of the FPGA design is dependent on the size of the target being tracked. Larger crop regions will take more time to transfer and more time to scale for the FPGA. In our experiments we tracked targets from 20×20 to 40×40 pixels without any significant change in performance. However in extreme cases, we expect this to have a measurable effect. The GPU implementation accepts the entire frame as input. This incurs more transfer time overhead, but has the added benefit of being shared between tracking targets.

Both implementations scale well with additional targets, and at nearly the same rate. For the same period of time, the total number of tracked targets is higher at the lower frame rate. The GPU implementation frame rate is however less than real time at

multiple points.

7.7 Conclusion

In this paper, we have addressed the task of accelerating an online boosting based multi-target tracker. We have proposed and evaluated two hardware designs: a GPU design and a FPGA design. Speed ups over the software-only C++ implementation are 2.7× for the GPU design and 68× for the FPGA design. The FPGA design is capable of tracking 57 independent targets at 30 FPS. The FPGA design also leverages pipelining and accelerates most of the algorithm which leads to better performance.

Acknowledgment

This chapter contains material printed in Field-Programmable Custom Computing Machines (FCCM), 2014. The chapter also contains material that was omitted from the publication due to space constraints. The work described in this chapter is a collaboration with Siddarth Sampangi. The dissertation author was the primary investigator and author of this paper.

Chapter 8

Improving FPGA Accelerated Tracking with Multiple Online Trained Classifiers

8.1 Introduction

Robust visual object tracking has improved considerably in recent years and is becoming a widely used enabling technology. It empowers a host of applications in fields such as: human computer interaction [9], autonomous vehicles [6], and video surveillance [54]. Yet it is still an especially difficult task for computers as objects change in appearance over time. Rotation, scaling, and using lighting insensitive color space transformations can compensate for some changes in appearance. But rotations out of plane, occlusions, and object deformation still present a problem for most algorithms.

Algorithms that learn an appearance model online have proven effective for robust tracking [31, 7]. Instead of training offline with large volumes of examples, the approach is to train online using a modest number of examples gathered at runtime. This approach has the benefit of training with examples that match the current appearance. But it requires additional online computation which can affect runtime performance.

Boosting for feature selection has been used with great success for detection [58] and tracking [31]. Boosting is the process of combining many weak classifiers

into a single strong classifier that performs better in aggregate. Training examples are used in an iterative process to identify the most discriminative weak classifiers. In the tracking context, weak classifiers are image features. Online boosting for tracking is an adaptation of boosting that fits the online training approach. Appearance model features are boosted to generate a classifier that can detect an object as it currently appears. As the object's appearance changes, new examples are gathered, and the classifier is updated accordingly.

Our contributions in this paper deal with online boosting for tracking. Tracking has three main components: an appearance model which identifies the object, a dynamics model which governs how the object moves, and a search strategy which defines where to look for the object. We address the challenge of tracking visibly changing objects through improvements in the appearance model. This work is based on our previous work in FPGA accelerated online boosting [40]. We improve upon the single online boosted tracker approach by using multiple classifiers learned at runtime. These classifiers are trained to recognize specific poses of a target which helps maintain location accuracy. To use multiple trackers effectively, we present a novel method for comparing classifier scores. Lastly, we accelerate our algorithm using a FPGA. The additional computation power from the FPGA allows the algorithm to evaluate and train up to 11 different classifiers each frame at 60 frames per second (FPS). Compared to the original algorithm, our work shows not only improvements in runtime performance, but in tracking accuracy as well. The contributions of this paper are:

- An algorithm for learning a pool of pose-specific classifiers at runtime.
- A method for comparing multiple classifier scores.
- A FPGA-CPU design and implementation of our algorithm for robust tracking.

The rest of the paper is organized as follows. We discuss related work next. The

algorithm is described in Section 8.3. The FPGA design is presented in Section 8.4. This is followed by experimental results in Section 8.5 and conclusions.

8.2 Related work

Research in object tracking frequently takes one of two directions. Research in tracking algorithms and features often leads to improvements in accuracy. While research in hardware acceleration often leads to improvements in runtime performance. In our work, we provide an algorithm and FPGA accelerated design that leads to improvements in both.

From an algorithmic perspective, our design uses an online boosting approach for the appearance model similar to that employed by Viola [59]. It builds on the work of Babenko [7] by using classifiers trained at runtime. Several other algorithms use this adaptive approach [32, 31, 1] including the well known Predator algorithm by Kalal[41]. These algorithms attempt to learn a representation of the target object using a single classifier over time. This can be a successful approach for certain classes of objects that have a limited number of visual representations. For deformable objects or objects that can vary substantially in appearance, many classifiers do not have the expressiveness to learn all representations. Our approach is unlike other adaptive appearance algorithms because we employ multiple classifiers to handle changes in appearance. Our tracking classifiers adapt to a short term appearance history. Our pose-specific classifiers detect previous representations with high discriminative capability. Both classifiers are used to track objects as they move and change in appearance. This approach requires incorporating the multiple classifier predictions into a single algorithmic prediction.

From a hardware perspective, our work with online boosting is similar to that contributed by Lo[45]. Lo proposes a method for accelerating boosted training for Viola-Jones style detectors. They achieve a 14$\times$ speed up over a CPU. But their work is aimed

Algorithm 4. Classifier Tracking Algorithm

Input: New image frame at time t

1: Select a set of samples, cropped from frame,
$X^s = \{x | s > \|l(x) - l^*_{t-1}\|\}$
2: Calculate Haar feature values, $f(x)$, for $x \in X^s$
3: Use classifier, $H(x) = \sum_k h_k(x)$, to classify samples X^s
4: Set new location $l^*_t = l(\text{argmax}_{x \in X^s} H(x))$
5: Select positive and negative samples sets from frame,
$X_1 = \{x | r > \|l(x) - l^*_{t-1}\|\}$
$X_0 = \{x | q \leq \|l(x) - l^*_{t-1}\| < q\prime\}$
6: Train classifier on positive and negative sets,
$H = Train(X_1, X_0)$

at accelerating traditional offline boosting. Other FPGA tracking accelerated work in the field focuses mostly on accelerating classifier evaluation [4] or search methods [16]. Most are standalone designs that do not improve upon the accuracy of the algorithms they accelerate. Moreover, to our knowledge, no other work accelerates online training using dedicated hardware.

8.3 Algorithm

Our algorithm uses multiple concurrent classifiers each frame and merges their outputs to select a new location.These classifiers follow an online boosting algorithm proposed by Babenko [7].

8.3.1 Classifier Algorithm

Babenko's algorithm is an online boosting algorithm for tracking which uses Multiple Instance Learning [22]. It was selected because of its adaptive appearance model. It uses a first order dynamics model for motion and uniform search strategy. The algorithm consists of two steps: find the new target location, then update the trained classifier. For each frame, the first step evaluates windows in a region surrounding the last known target location (radius s). Evaluation yields a new location. Then the region

surrounding the new location is used to to select positive and negative training examples (radii q and $q\prime$). These examples are used to incrementally train the classifier for the next frame. This tracking flow is illustrated in Algorithm 4.

The algorithm uses a boosted collection of Haar-like features for the appearance model. A Haar-like feature is a collection of two to six weighted rectangular regions defined within a window.The rectangles define regions of the window over which to sum pixel values. The rectangles need not be adjacent. We refer to these features simply as Haar features in the rest of this paper.

Boosting is an approach which combines several weak classifiers, h, into a strong classifier, H by iteratively maximizing the log likelihood of the strong classifier $\log \mathscr{L}(H_{k-1}+h)$. At each iteration, k, the existing strong classifier is combined with the next weak classifier, h, that maximizes this quantity over the training data.

At initialization, a pool of M Haar features are generated with random rectangle coordinates and weights. During training, K of these features are selected and are used as weak classifiers, $h(x)$, to form the strong classifier, $H(x) = \sum_k \mathbf{h}_k(x)$. Each weak classifier consists of a Haar feature and four additional parameters: μ_1, σ_1, μ_0, σ_0. These parameters quantify the distribution of Haar scores, $f(x)$, for positive and negative examples respectively. They are updated during training using examples collected from the current frame. At runtime, each weak classifier calculates the score of a sample window, x, using the log odds ratio:

$$\begin{aligned} h(x) = &\frac{1}{2\sigma_0}(f(x)-\mu_0)^2 - \log(\frac{1}{\sqrt{\sigma}_0}) \\ &-\frac{1}{2\sigma_1}(f(x)-\mu_1)^2 + \log(\frac{1}{\sqrt{\sigma}_1}). \end{aligned} \tag{8.1}$$

This formula results in higher scores the closer a sample's feature value is to the positive mean and the further it is from the negative mean.

Algorithm 5. Main Tracking Algorithm

Input: New image frame at time t

Evaluate all classifiers at location
$\{X^1, X^2, X^3, X^{Pool}\} = Evaluate(T^1, T^2, T^3, Pool)$
Calculate majority, $x_T^* = Majority(X^1, X^2, X^3)$
Calculate pose-specific classifier pool maximum,
$x_{Pool}^* = \text{argmax}_{x \in X^{Pool}}(Kurtosis(x) \times Score(x))$
Use maximum as location, $l^* = l(\text{argmax}_{x_T^*, x_{Pool}^*} Score(x))$
Identify pool classifiers with detections,
$D = \{p \in Pool | ValidDetection(p)\}$
Update least used priority queue, $Q = UpdateQueue(Q, D)$
Start training a new pose-specific classifier if possible,
if $Stable(X^1, X^2, X^3)$ and $P_{new} == none$ **then**
    $P_{new} = \min(Q)$
    $Pool = Pool \cup P_{new}$
else if not $DetectedEnough(P_{new})$ **then**
    $Pool = Pool \setminus P_{new}$
    $P_{new} = none$
end if
Train classifiers, $Train(T^1, T^2, T^3, D)$

Each classifier employed by our algorithm uses a modified version of this tracking algorithm. To improve tracking efficacy, we change the classifier's scoring function to only use the positive example distribution:

$$h(x) = -\frac{1}{2\sigma_1}(f(x) - \mu_1)^2. \tag{8.2}$$

Positive training examples gathered across frames will likely have a similar distribution because they are all sampled from the target's location. However, negative examples are drawn from multiple different locations in a frame and across frames. A single normal distribution will model this distribution poorly and distort scores. In practice, we confirmed that using our scoring function produced more reliable, more stable scores. This does not mean negative examples are ignored. Negative examples are still used during classifier training.

Each classifier is also modified to calculate normalized cross-correlation scores for

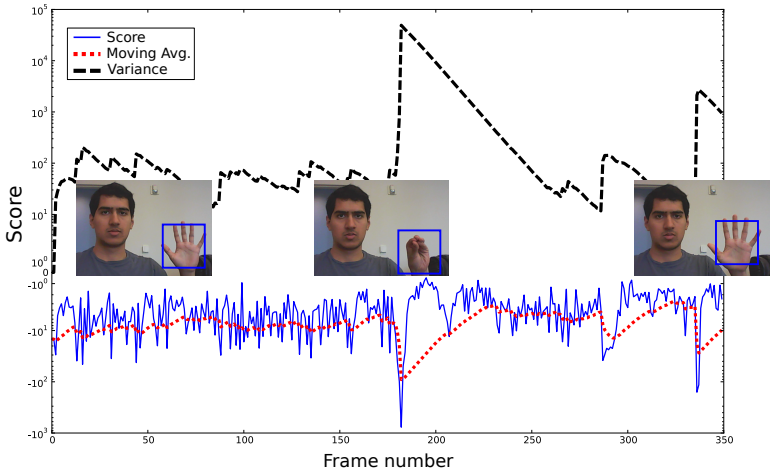


Figure 8.1. Classifier scores during target appearance changes. Changes produce sharp drops in score and spikes in variance.

each window, in addition to Haar feature based scores. The normalized cross-correlation is computed between a pixel template captured at runtime and each frame window in the search region. Both Haar and normalized cross-correlation are used as independent methods of locating the target object within the search region.

Lastly, each classifier calculates the mean and variance of scores across all windows in the search region. It also returns the 128 highest scoring locations detected in each frame, instead of just the maximum location.

8.3.2 Main Algorithm

Our algorithm is both motivated and enabled by FPGA acceleration. FPGA acceleration provides runtime resources beyond a typical CPU based implementation. These additional resources can be used to track multiple independent targets through simple replication. They can also be used to improve tracking a single target. However, combining the results of multiple classifiers is not as straightforward. Our algorithm uses multiple classifiers per target in two different ways. It uses three adaptive classifiers for a single target, and trains pose-specific classifiers to be evaluated concurrently each frame.

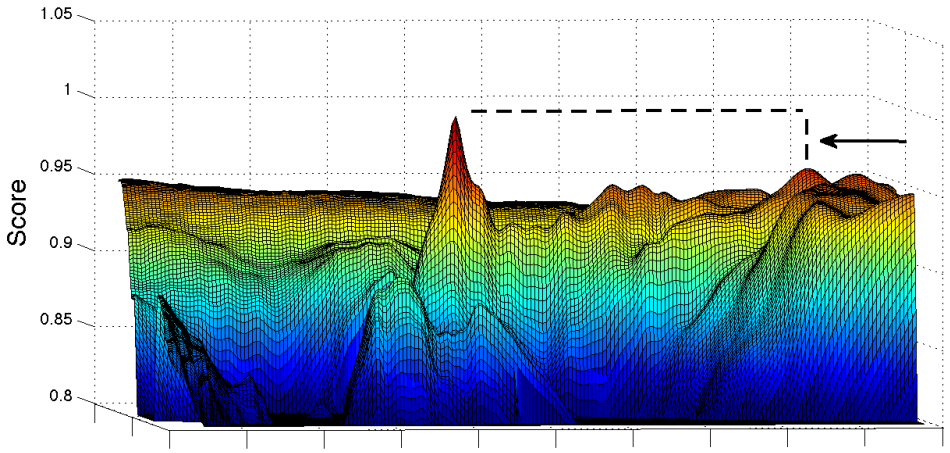


Figure 8.2. Plot of a classifier's score over the X and Y dimensions. This example shows a sharp peak and the next highest peak at least d pixels away. The variance normalized difference between these two peaks is the kurtosis. Larger differences higher confidence in the maximum location.

This flow is described in Algorithm 5.

Our algorithm uses three adaptive classifiers with a majority aggregation function. These classifiers are evaluated and trained every frame. The majority location is the new target location. Any adaptive tracking classifier that drifts over a threshold distance away from this location is re-centered on the majority location. Using multiple classifiers improves robustness because each classifier is able to track a different part of the target, at a different scale, with different features. We found using the majority provided better results over using the maximum because spurious, high scoring, incorrect maximal locations from a single classifier do not influence the majority location.

Before using multiple classifiers, we attempted to mitigate target loss by using more features with a single tracker. This approach showed improvement, but did not reduce drift or target loss by any significant amount. The number of features that can fit in a fixed sized sliding window are limited. Adding additional features per classifier has a diminishing margin of return but reduces runtime performance linearly.

Our algorithm also learns pose-specific classifiers at runtime and evaluates them

concurrently with the adaptive tracking classifiers. Pose-specific classifiers are trained to detect a specific representation of the target. The intuition is that a single tracking classifier cannot effectively learn every representation of an object (imagine a vehicle from different angles or a human hand performing gestures). Individual classifiers can however learn a specific object pose with high discriminative ability. If such poses are revisited in future frames, trained pose-specific classifiers can be used to recover from tracking drift or even complete target loss.

Unlike the adaptive tracking classifiers, pose-specific classifiers are trained only when the target appears in the same pose. We take an active learning approach to detect these situations. For each frame, if the pose-specific classifier's maximum scored location and a normalized cross-correlation maximum location agree, the detection is valid and the classifier is trained. Otherwise, no training takes place. Valid detections are used as potential new target locations. The normalized cross-correlation is performed using a template acquired on the first frame of training.

Creating a pose-specific classifier risks training on something other than the target. This can happen when the current location has drifted or the target is lost. The algorithm cannot detect drift or loss without supervision or feedback. Instead it waits until the tracking classifiers are temporarily stable. Stable simply means the target is currently being tracked (moving or not) with high confidence. Our algorithm identifies these times using the current score, average score and variance. Times of instability are punctuated by large drops in score, followed by a period of adaptation characterized by high variance (see Figure 8.1). Constant training makes it impossible to learn a consistent mean, so a decaying average mean is maintained, along with its variance. When the current score is above the mean and the variance has decreased below a threshold, the tracking classifiers are considered temporarily stable.

Only one pose-specific classifier is trained at a time. It is given a fixed number of

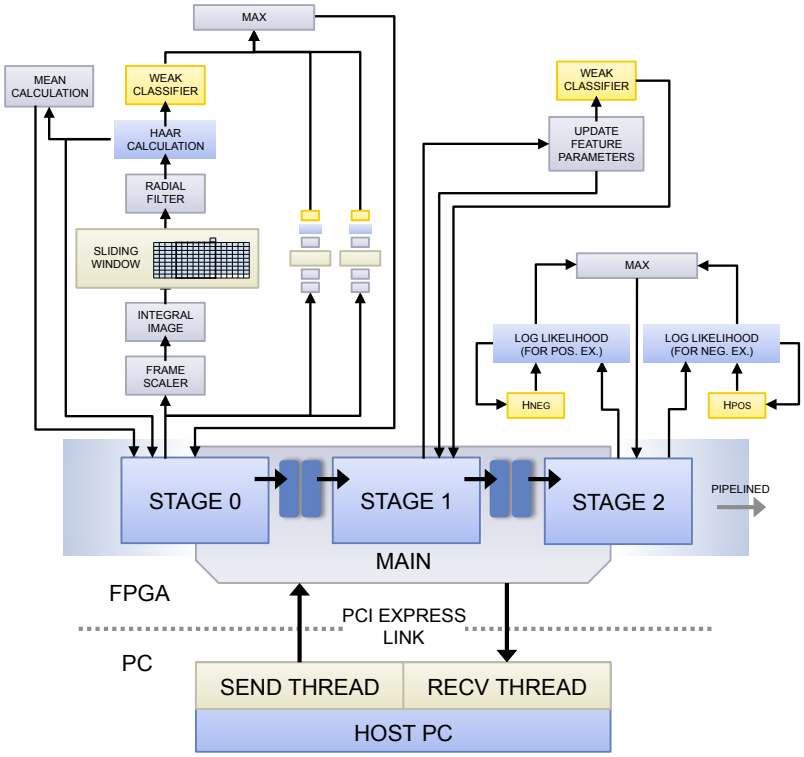


Figure 8.3. FPGA-CPU high level architecture. The FPGA design is composed of a three stage pipeline: Evaluate, Update, and Train. The pipeline is controlled by a state machine that also interfaces with software running on the host PC.

frames within which it must be trained a minimum number of times. If the pose does not exist long enough, the classifier will not meet the minimum training threshold and it is discarded. If it does meet the threshold, it is kept and added to a pool of concurrently evaluated pose-specific classifiers. Each frame, all classifiers in the pool are evaluated at the current location.

Each additional pose-specific classifier consumes runtime resources. For a given performance level, only a fixed number can be evaluated at a time. But video sequences may produce an unbounded number of new pose-specific classifiers. We therefore, employ a least used eviction model to limit the number of classifiers evaluated each frame. The most used classifiers are prioritized highest in a queue. The lowest in this queue is replaced when a new classifier starts training. Detections by a classifier increases

its queue priority independent of other classifier detections.

Lastly, evaluating multiple pose-specific classifiers requires an aggregation function when multiple valid detections occur. Our algorithm uses the maximum classifier score multiplied by its kurtosis. The kurtosis provides a measure of the score distribution's "peakedness". The algorithm calculates kurtosis over the distribution as:

$$\frac{\max\{h(x)\} - \max\{h(x)|d > \|l(x) - l^*\|\}}{\sigma_{h(x)}}. \tag{8.3}$$

This calculates the difference between the global maximum score and a local maximum score, normalized by the standard deviation of all scores, $\sigma_{h(x)}$. The local maximum score is the largest score at least d pixels away from the location of the global maximum l^*. The d pixel radius is selected to separate independent peaks. A small difference between maximums suggests low confidence in the global maximum location, as the next highest peak is also a good candidate. Figure 8.2 illustrates this measurement graphically with a high confidence example.

8.4 FPGA-CPU design

The FPGA-CPU design is a partitioned application that accelerates boosted classifier evaluation and classifier training. The tracking application runs in software on the CPU. However, all of the feature extraction, evaluation, and training take place on the FPGA. Only image cropping, rendering, and various bookkeeping tasks run on the CPU.

Each video frame is processed by the algorithm in two passes. The first pass performs the target search by evaluating classifiers in a sliding window fashion over a region of the frame. This search limits evaluation to windows within a runtime specified radius of the last known location. The search is a dense evaluation of all possible (overlapping) windows. After the search identifies the new target location, the classifiers

are retrained. This takes place during the second pass. Training examples are sampled from the current frame at the newly identified target location. All windows in a small radius at the new location are used as positive examples. Negative examples are sparsely sampled from a distantly spaced annular region surrounding the new location. This process is described in more detail in previous work [40].

The high level architecture of this design is illustrated in Figure 8.3. The CPU communicates with the FPGA over a PCIe link using the RIFFA framework[39]. CPU software invokes the IP cores on the FPGA with input data and receives the response as output. The FPGA design consists primarily of a three stage pipeline. The *Evaluate* stage evaluates trained classifiers on the video frame. The *Update* stage uses the output of the Evaluate stage to update classifier parameters. These updated parameters provide the basis for a new classifier. The *Train* stage selects which features, and thus which updated parameters, are part of the new classifier.

8.4.1 Evaluate stage

The Evaluate stage is responsible for extracting Haar features, calculating classifier scores and normalized cross correlation on windows within the frame. The architecture of this stage is illustrated in Figure 8.4. Input frame pixels, an image template, and other parameters are supplied via the PCIe link. Parameters and template pixels are stored in Block RAM (BRAM). Frame pixels are processed as they are streamed through the data path.

Input frame pixels are first rescaled to fit within a fixed size window in a sliding window pipeline. In our previous work accelerating online boosting for tracking, we used a 20×20 pixel window of registers. This size was selected based on numerous other publications using similar sized sliding window designs. In practice however, we found that many of the targets we tracked required considerable down sampling to fit

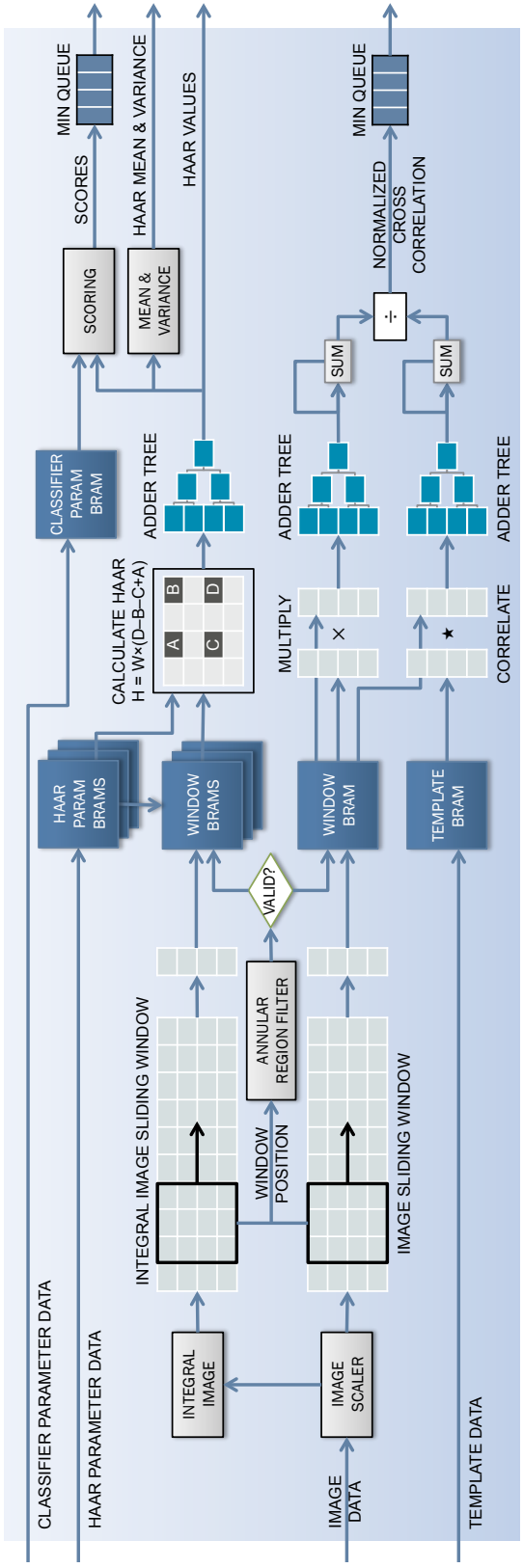


Figure 8.4. Evaluate stage architecture. Input pixels are scaled and then converted into integral image format. Two parallel sliding window pipelines calculate Haar and NCC values. Haar values are scored according to weak classifier parameters. The score mean and variance are calculated incrementally. Both the scores and NCC values are added to a priority queue. The priority queue captures the top 128 maximum values across the frame. These values are outputted as the result.

in these dimensions. This resulted in loss of texture and detail in the image. Because Haar features work best with high frequency texture and sharp edges, this introduced a source of error for our calculations. We expect this is a common source of error for many sliding window designs, yet it is often unaddressed in the literature. In this work we use a 45×45 pixel window, backed by BRAM. Experiments with the two sizes confirmed an improvement in tracking accuracy with the larger window size. Using BRAM instead of registers requires multiple BRAM modules to maintain parallel access to the window data. However an equivalent sized register backed window would exceed the look up table (LUT) capacity of most commercially available FPGAs. This is due to the quadratic growth of register and MUX usage with size.

The image scaler uses bilinear interpolation to scale. Four weighted pixels are used to generate a single output pixel. This provides arbitrary precision output scaling. The scaled pixel data is converted into integral image format and both streams are passed to parallel sliding window buffers. During conversion, only the least significant 19 bits of each integral image pixel are retained. This does not affect Haar feature calculation as higher order bits will be subtracted away during calculation. The two sliding window buffers move in unison to maintain the same window position across the frame. The image sliding window provides data for normalized cross correlation (NCC) calculation. The integral image sliding window does the same for Haar feature extraction and classifier scoring. An annular region filter determines if the current window is within a circular radius or annual region, for both calculations.

The sliding window buffers provide a new column of 45 pixels every cycle. For both calculation data paths, each new column of pixel data is buffered into a BRAM with a 45 pixel wide data port. After 45 cycles, an entire window has been buffered. Each subsequent cycle provides a new column of pixels and results in a new window. Column data is stored in circular manner in the window BRAMs.

For each newly buffered window, the NCC calculation processes the 45 saved columns, one column of per cycle. It squares the column data and cross correlates it with a column of template pixel data. Template data has the same dimensions as a window and is accessed from a separate BRAM. This is done with 45 parallel modules. The results are summed in adder trees and then added to separate running sums. It requires 45 cycles to access all columns of the data and process a single window. The NCC values are then streamed to a priority queue.

Concurrently, Haar feature values are computed in the feature scoring computation data path. Six parallel modules access Haar rectangle coordinates from independent window data and parameter BRAMs. This allows a new Haar value to be calculated each cycle with up to six rectangles per feature. The Haar rectangles are weighted and summed via an adder tree and the resultant Haar feature values are simultaneously outputted and streamed to the Mean & Variance and the Scoring module. The design supports up to 64 Haar features during evaluation. We used 50 features in our experiments. Therefore, it takes 50 cycles to calculate all the Haar features for each window.

The Mean & Variance module calculates and outputs the $E(x)$ and $E(x^2)$ values for each Haar feature across all windows. The Scoring module uses the Haar feature values and classifier parameters to score each window according to Equation 8.2. These window scores are then streamed to a priority queue. Both modules use floating point operators to preserve precision over the wide range of values each output takes. The algorithm is also sensitive to small changes in value. A fixed point data path with sufficient accuracy to accommodate output values would require an unreasonably large number of bits.

The priority queues for both data paths are minimum priority queues, 128 values deep. They keep the NCC and score values in a partial sorted order. The minimum value is always maintained. Values are paired with the window position to which they

correspond. New values are added to the queue as they are generated. After the queue fills, the lowest value is removed to make room for the next new value. At the end of processing all windows, the queues hold the top 128 maximum values. The queues are then drained and outputted in increasing value order.

8.4.2 Update stage

The Update stage uses the the Haar feature means of the positive and negative examples, calculated during the Evaluate stage, to update the current classifier parameters. These new parameters define the weak classifiers that can be boosted during training into the next classifier. After each parameter is updated, it is used to score the Haar feature values from the Evaluate stage. This requires iterating over the Haar features in feature major order instead of window major order as they were generated. Scoring performs the same calculations as in the Evaluate stage, but uses the updated parameters. The parameters are outputted and the window scores are stored for the Train stage. Figure 8.5 illustrates this process.

8.4.3 Train stage

The Train stage calculates the log likelihood of each positive and negative example window using the scores from the Update stage. Log likelihoods for each weak classifier are summed over examples. Then the weak classifier which contributes the minimum negative log likelihood across all examples is added to the classifier. This is an iterative process that evaluates 256 weak classifiers to select 50 for inclusion in the classifier. The log likelihood calculation is performed by 16 parallel modules that operate on positive and negative examples. This parallelization factor was selected to balance the runtime between stages. Floating point operators are used for the exponentiation, logarithm, and division in this calculation. After each iteration, the selected weak classifier id is

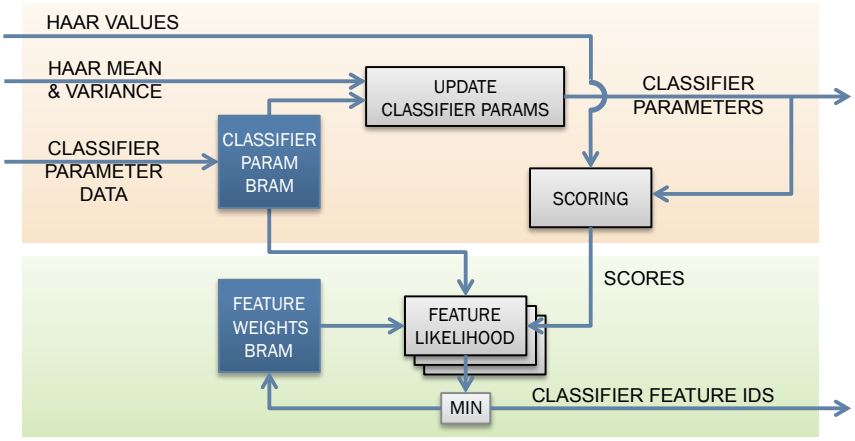


Figure 8.5. Update stage (top) and Train stage (bottom) architectures. The Update stage updates classifier parameters using the positive and negative examples. It also scores all examples using the updated parameters. The Train stage uses the scored examples to iteratively select the best weak classifiers.

outputted and the intermediate classifier is updated. Figure 8.5 illustrates the high level architecture. Further details on the Train and the Update stages can be found in our previous work [40].

8.5 Experimental results

The FPGA-CPU design is implemented in Verilog on a Xilinx Virtex 7 VC707 using Vivado 2013.3. It runs at a frequency of 250 MHz. It is connected to a 4 core Intel i7 3.6 GHz system with 16 GB RAM via a x8 PCIe Gen 2 slot. The portion of the algorithm running on the CPU is written in C++. The software primarily renders video and performs bookkeeping tasks. Multiple threads are used to coordinate communication with the FPGA. The FPGA design resource utilization is listed in Table 8.1

A software-only implementation of our algorithm was also written in C++ and run on the same computer. It is multi-threaded with the OpenMP library and uses Intel Integrated Primitives vector instructions. This highly optimized software-only implementation can run a single tracker at 17 FPS. Our algorithm uses three trackers

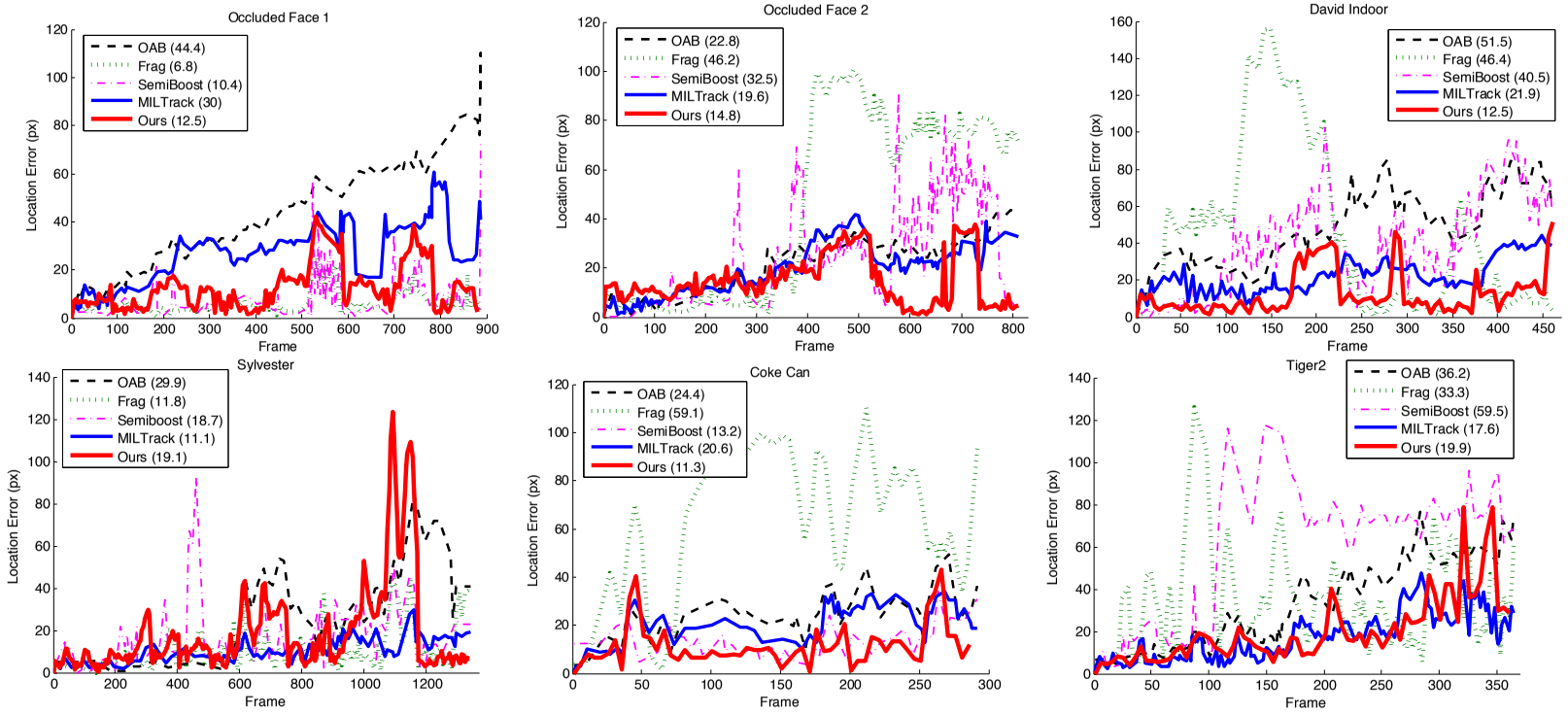


Figure 8.6. Location errors on video sequence from several tracking publications. Error is the difference between predicted tracking location and the ground truth, in pixels. Average pixel error over the entire sequence is shown in parentheses. Algorithms compared include: Online Adaboost (OAB) [31], FragTrack (Frag) [1], Semi-supervised Online Boosting (SemiBoost) [32], and MILBoost Tracker (MILTrack) [7]. Video sources are: Occluded Face 1 [1], Occluded Face 2 [7], David Indoor [53], Sylvester [53], Coke Can [7], and Tiger2 [7].

along with up to 7 additional classifiers each frame. When running the full algorithm, the software-only implementation runs at 2 FPS. The FPGA-CPU implementation runs the full algorithm with all trackers and classifiers at 60 FPS. This represents a 30× speed up over the software-only version.

To evaluate the accuracy of the FPGA design, we tested the implementation using a set of standard tracking videos from recent publications [1, 53, 7]. Tracker error is measured in terms of distance (in pixels) between the center pixel and a manually determined ground truth. The average pixel error for the entire sequence is provided in parentheses. We compare our algorithm against that proposed by Babenko (MILTrack) as well as several other state of the art algorithms. Figures 8.6[a-f] show the error plots.

The plots show that our algorithm performs very well on the Occluded Face, David, and Coke Can video sequences. The Occluded Face sequences track a face while it is repeatedly occluded and revealed. During occlusion, the tracking classifiers inevitably update to track features on the occluding surface. Each time the face is revealed, these classifiers drift off the face. The error plot shows this behavior is consistent across all the tracking algorithms. However, our pose-specific classifiers detect the face location as soon as it is revealed and quickly re-centers the tracker.

In the David video sequence, the target is again a face. But this face rotates in and out of plane and the scene lighting changes considerably. Additionally, eyeglasses are present at the beginning of the sequence and are later removed. Again, the tracking classifiers perform well but are prone to drift and loss. Many different poses of the face are learned during the sequence. This helps recover from loss several times, as illustrated by the sharp drops in error.

The Coke Can video tracks a hand held can of soda as it is spun, rotated, moved between different lighting conditions, and behind house plants. The rigid shape of the can provides a good tracking feature for all the algorithms. Pose-specific classifiers quickly

Table 8.1. FPGA design resource and VC707 utilization.

Slice Reg.	Slice LUT	BRAM	DSP48E
181541	135169	520	320
30%	45%	50%	11%

learn the few poses the target can take and provide quick recovery after occlusion or temporary changes in appearance. This recovery improves the accuracy considerably over the MILTrack algorithm.

In the Tiger2 video sequence, the performance is about the same as the MILTrack algorithm. This sequence tracks a stuffed animal tiger as it's moved in and around plant foliage. The target is occluded quite a bit of the time by different plant leaves and the motion is fast and erratic. As a result, no pose-specific classifiers are given the opportunity to train. Even when the training threshold is reduced, the plant occlusion makes it extremely difficult to detect similar poses in the sequence. Thus, the performance of our algorithm is on par with MILTrack.

The Sylvester sequence exhibits target loss during tracking and highlights a weakness in our algorithm. The Sylvester sequence tracks a stuffed cat as it is articulated and flown around a room. The scene is cluttered and has extremely high lighting variation. The motion is rapid, but there is occasionally enough stability for pose-specific classifiers to train. This provides recovery several times in the sequence. However the back and forth motion challenges the tracking classifiers. The majority aggregation function for the tracking classifiers suppresses spurious high scoring incorrect detections. But this can impose a lag when reacting to motion as the majority of the classifiers must agree to the change. In this video sequence, this lag results in drift and then loss. It should be noted, that a pose-specific classifier recovers from the target loss before the end of the sequence.

Across all but one of the video sequences, our algorithm performs equal to or significantly better in terms of average accuracy. These sequences come from several

different tracking publications. They were selected based on the availability of results from existing algorithms, not based on the performance of our algorithm. They are difficult sequences representative of many appearance changing scenarios.

8.6 Conclusion

We have provided a FPGA-CPU accelerated design for tracking objects through appearance changes, using multiple online boosted classifiers. The design accelerates our algorithm for learning a pool of pose-specific and tracking classifiers at runtime. It also employs a novel method for comparing multiple classifier scores using a kurtosis of the score distributions. Compared to a multi-threaded software-only CPU based implementation, it boasts a 30$\times$ speed up. Our algorithm performs at state of the art levels and shows an improvement in accuracy over existing tracking algorithms.

Acknowledgment

This chapter contains material as it appears in Field Programmable Logic and Applications (FPL), 2014. The work described in this chapter is a collaboration with Siddarth Sampangi and Yoav Freund. The dissertation author was the primary investigator and author of these papers.

Chapter 9

Future Directions

Using the Smart Frame Grabber Framework over several projects helped us refine the details. There have been four releases of the RIFFA framework as of this writing. It also gave us a perspective on computer vision application acceleration. Using FPGAs to accelerate computer vision applications is still a difficult task undertaken by those who are willing to invest considerable time and effort to learn the tools. Using frameworks like the Smart Frame Grabber can alleviate some of this difficulty, but taking advantage of heterogenous platforms can be made easier. Below we identify some future directions for exploration.

9.1 Simple Compatible Interfaces

There is no common interface that one can put on every component. For example a sliding window pipeline needs signaling that color space converter does not. However, we found component reuse is best achieved by establishing simple, generally applicable interfaces for each component. Erring on the side of simplicity encourages reuse. Reducing the amount of understanding necessary to use a component greatly reduces development time and problems due to misconfiguration.

On the surface, using simpler interfaces has a downside. It would seem that one cannot configure the component as needed for every possible situation. Given the

complexity of most components we have seen, this must be a chief concern among designers. However this can easily be solved by having a complex interface driven by an adapter with a simple interface. This type of approach will not compromise performance or flexility. Moreover, a ready-to-use simple component will be used more often than a ready-to-configure component.

There has been a recent push by Xilinx to use the AXI bus specification as the common interface for data on all of Xilinx's IP cores. This is a promising direction as it includes flow control and does not preclude the use of additional control signals.

9.2 Direct Device To Device Communication

RIFFA has made integration of CPUs, FPGAs, and GPUs an easy to achieve prospect. There is demand however for direct device to device communication. For high performing applications, using shared CPU memory as the method of passing data between devices can be too slow. Allowing FPGAs and GPUs to send and receive data directly can have a significant performance impact.

The PCIe architecture already supports this type of direct communication. However, it is not often exploited as most commercial uses of PCIe are for single device accelerators.

9.3 OpenCV Integration

OpenCV is a popular software library for computer vision programming. It supports multiple CPU architectures and implements many computer vision algorithms. It has gradually been including support for GPU based acceleration. This is a good step towards wider adoption of hardware accelerated applications. Including FPGA based acceleration, perhaps via RIFFA, would be another great step. Doing so would require establishing a common interface for input and output. But the rest of the pipeline would

be free for implementation. As many computer vision applications start with the OpenCV libraries, extending it to include multiple device accelerators is a natural choice.

Appendices

Appendix A

RIFFA 1.0

A.1 Getting Started

The RIFFA distribution should be downloaded from the Downloads page on the RIFFA website. After saving the distribution to disk, open it up and read the README.txt and release notes. The distribution will contain RIFFA IP cores, the Linux kernel driver, and a C/C++ user library. It will also contain example designs with IP cores and user applications. You should look at the example designs to see how the system is configured. The follow instructions are designed to help you integrate RIFFA into an existing design.

Xilinx instructions for PLB based systems:

1. Add the RIFFA distribution pcores into either your XPS global pcore repository or into the pcores directory of your base system directory. You may need to restart XPS for these changes to take affect.

2. Add a plbv46_pcie Xilinx PCIe Bridge core and xps_central_dma Xilinx DMA core to your design.

3. Connect the master and slave ports of the xps_central_dma and plbv46_pcie to your PLB system bus.

4. Configure the xps_central_dma with:

(a) The maximum FIFO depth (currently 48).

(b) The maximum read & write burst size (currently 16).

(c) Make sure it has a valid address on the system bus. This address will be used in step 9.

5. Configure the plbv46_pcie with:

(a) A vendor id = 10EE (for Xilinx) and a device id appropriate to your FPGA (the ML505 = 0505). These values will be used by the driver install below.

(b) The completion timeout should be checked.

(c) The PCIe capabilities register slot implemented should be checked.

(d) Configure 1 PCIe BAR, of size 213 = 8 KB. The address should be 0x80000000. This value will be used in step 8.

(e) Configure 6 IPIF BARs, of size 222 = 4 MB. The address of the first IPIF BAR should be 0xA0000000. All the bars should follow and be adjacent (e.g. the second IPIF BAR starts at 0xA0400000). The remote translation address can be left at 0x00000000. These values will be assigned at boot time by the PC. The starting address and size of IPIF BARs is used by the driver.

(f) Make sure the plbv46_pcie has a valid address on the system bus. Note, this is separate from the BAR addresses. In the Address tab, this shows as the C_BASEADDR address for the plbv46_pcie. This value will be used in step 9.

6. Add a central_notifier, a simbpus_mst_plbv46_adapter, and a simpbus_slv_plbv46_adapter to the design.

7. Connect both adapter cores to the system bus and connect the SIMPBUS interfaces on both adapter cores to the central_notifier.

8. Configure the simpbus_slv_plbv46_adapter to have an address of 0x80000000 and size of 8 KB. This must be the same address and size as in step 4. In truth, the address can be anything, as long as it doesn't conflict with any other core on the system bus and is consistent with the value of the PCIe BAR.

9. Configure the central_notifier with:

   (a) Set the number of channels as desired. This enables the same number of buses on the Bus Interfaces tab.

   (b) Check the init bus box if you want to enable and connect an IP core to perform system initialization using the SIMPBUS interface. Note, do not check this box unless you do connect an IP core that will perform initialization.

   (c) Set the DMA address to the value assigned in step 4.3.

   (d) Set the PCIe Bridge address to the value assigned in step 5.6. This is the C_BASEADDR value of the plbv46_pcie component.

   (e) Set the IPIF BAR length to 4 MB.

   (f) Configure the SIMPBUS data width to the PLB data width.

10. Connect the remaining central_notifier ports:

    (a) SYS_CLK should be connected to the clock used for the PLB bus.

    (b) SYS_RST should be connected to the system reset.

    (c) INTR_PCI should be connected to plbv46_pcie's MSI_request port.

    (d) INTR_DMA should be connected to the xps_central_dma's IP2INTC_Irpt port.

    You can now connect your custom IP core to the central_notifier's RIFFA channel

interface on the Bus Interfaces tab. An example IP core that uses the RIFFA channel interface is provided in the examples directory of the RIFFA distribution.

The Linux driver must be installed before applications can access the FPGA. The driver is located in the central_notifier's pcore directory, under sw/linux/driver. To build and install the driver, you'll need root/sudo privileges. The steps are below:

Ubuntu/Debian and Fedora instructions:

1. In a terminal, move into the sw/linux/driver directory.
2. Execute: sudo make setup
    - This will ensure that your Linux system has the kernel headers that correspond to the current version of the kernel you're running. If you don't have them installed, this command will attempt to install them, typically into /usr/src. You only ever need to run this once to ensure you have the kernel headers. If you know you have them, you can skip this step.
3. Execute: make
    - This will compile the driver against your kernel. You will probably receive an error message indicating that you must specify variable value for VENDOR_ID and DEVICE_ID. You may update the Makefile to includes these values (see the top of the Makefile). Or you can pass them in via the command line (e.g. make VENDOR_ID=10EE DEVICE_ID=0506). These values correspond to the PCIe header values configured in your PCIe endpoint (see above). They must match or the OS will not load the driver when it detects the FPGA's PCIe endpoint.
4. Execute: sudo make install

- This will install the driver into the kernel to be automatically loaded at boot time. You can uninstall the driver by executing: sudo make uninstall. This will remove everything that was installed. If you want to manually load the driver you can always execute: sudo make load. As of this writing however, you'll need to reboot the computer every time you download a new image to your FPGA, so installing it usually the best choice.

5. Reboot.

    - After rebooting, log in and check for evidence that the driver loaded by executing: dmesg. If the driver loaded correctly you should see something like the following:

```
[   12.759709] FPGA PCIe endpoint name: 0000:02:00.00
[   12.759735] BAR 0 address: d0000000
[   12.759736] BAR 0 length: 8192
[   12.759791] fpga 0000:02:00.0: irq 28 for MSI/MSI-X
[   12.759819] MSI setup on irq 28
[   12.761444] gDMABuffer 0: ffff880037400000 -> 0000000037400000
[   12.763685] gDMABuffer 1: ffff8800bf800000 -> 00000000bf800000
[   12.765908] gDMABuffer 2: ffff8800bf400000 -> 00000000bf400000
[   12.768100] gDMABuffer 3: ffff8800bf000000 -> 00000000bf000000
[   12.770268] gDMABuffer 4: ffff8800bec00000 -> 00000000bec00000
[   12.772382] gDMABuffer 5: ffff8800be800000 -> 00000000be800000
```

    - If you don't see something like that, grep your /var/log/syslog for the term: FPGA. Double check that you've set the same VENDOR_ID and DEVICE_ID in your driver as you have in your PCIe endpoint configuration.

    - You may also check the /dev directory to see if the fpga device was created. It should be listed as /dev/fpga.

After you've synthesized your design, downloaded it onto your FPGA board, installed the driver, and rebooted your computer. You'll want to interact with the FPGA

from user software. You can find an example C application in the examples directory of the RIFFA distribution. It is located in the sw/linux/testapp directory of the riffa_example pcore. You may want to use this example as a template or model for your own application. Note that the fpga_comm.c/fpga_comm.h files in the sw/linux/testapp directory are the same as those in the sw/linux/userlib directory of the central_notifier pcore. The difference between these directories is that the Makefile in the sw/linux/testapp directory is designed to produce an executable whereas the Makefile in the sw/linux/userlib directory only creates object files. You may want to study the fpga_comm.h file as it contains the API for accessing the FPGA.

The basic pattern for accessing the FPGA via the RIFFA userspace API is as follows:

```
#define DATA_SIZE (8192)

int main(int argc, char* argv[]) {
 fpga_dev * fpgaDev;
 int recv, channel, timeout;
 unsigned int arg0, arg1;
 unsigned char data[DATA_SIZE];

 timeout = 10*1000; // 10 secs.
 channel = 0;
 arg0 = (unsigned int)rand(); // Random
 arg1 = (unsigned int)rand(); // values

 // Initialize the FPGA & open the channel.
 fpga_init(&fpgaDev);
 fpga_channel_open(fpgaDev, channel, timeout);

 // Send 2 arguments to the core on the channel,
```

```
  // then send a 'start' doorbell.
  fpga_send_args(fpgaDev, channel, arg0, arg1, 2, 1);

  // Receive the response data.
  recv = fpga_recv_data(fpgaDev, channel, data, DATA_SIZE);
  printf("Received data response, length: 0x%x\n", recv);

  // Close the channel and free the device.
  fpga_channel_close(fpgaDev, 0);
  fpga_free(fpgaDev);

  return 0;
}
```

In the example above, the statements in bold are the important statements. The program first initializes the fpga_dev pointer. Then opens a channel. It sends args to the IP core on the channel and starts the core by sending a doorbell. The program then waits for data to be returned on the channel. Up to 8 KB of data will be written to the data array. Any additional data received will be discarded. The amount of data received will be returned in the recv variable. The program then closes the channel and frees the fpga_dev memory. This example is fairly simple and meant to convey the usage model. All error handling code has been removed to preserve clarity. Your program should examine the return values for these functions and check for errors. The system log (or dmesg) can show errors encountered in the driver as well.

A.2 Hardware Interface

The RIFFA channel has several ports that are used for signaling between the PC and the FPGA. We've tried to keep it as simple and flexible as possible.

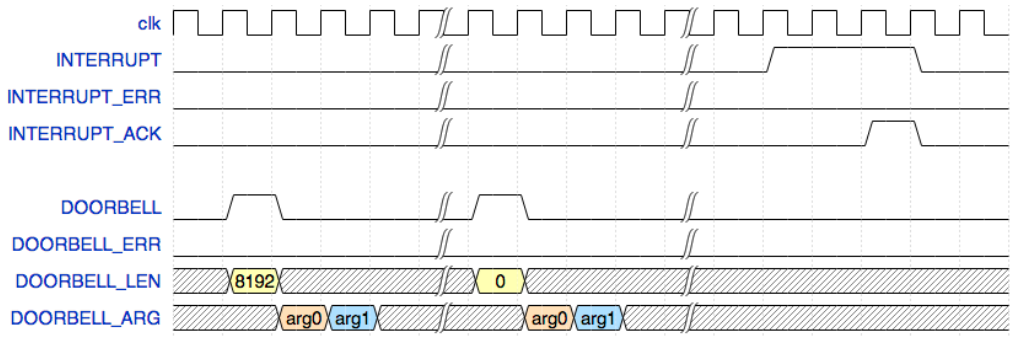


Figure A.1. RIFFA 1.0 timing diagram for doorbells/interrupts.

The RIFFA interface is a collection of ports for signaling data transfer, requesting buffers, and signaling events. Table A.1 describes the ports. The input/output designations are from the IP core's perspective.

Timing diagrams for common scenarios follow (again from the perspective of a connected IP core).

The timing diagram in Figure A.1 shows an IP core receiving a data transfer of 8 KB followed by a "start" doorbell signal (zero length). After some processing the IP core issues an interrupt, signaling completion. If the INTERRUPT_ERR port were high when INTERRUPT was asserted, this would signal an error condition for the interrupt. The INTERRUPT port must be asserted and held high until the INTERRUPT_ACK is pulsed. When a doorbell is received, the DOORBELL signal is pulsed at the same time the DOORBELL_LEN is valid. If the DOORBELL_ERR port were high when the DOORBELL port is pulsed, that would indicate an error condition for the doorbell. In the following two cycles, the DOORBELL_ARG port will output the "function call" style 32 bit arguments, arg0 then arg1. This pattern occurs every time a doorbell is received, regardless of whether the length and arg values are non-zero. Note that the arg values outputted are the last arg values specified by the PC. So they may be from previous "function calls".

Figure A.2 shows the timing for a DMA transfer from src_addr to dst_addr of

Table A.1. RIFFA 1.0 hardware interface.

Signal Name	I/O	Description
INTERRUPT	O	Assert high to signal an interrupt to the PC on the channel.
INTERRUPT_ERR	O	Assert high to signal an error with the interrupt. Only valid/used when INTERRUPT is high.
INTERRUPT_ACK	I	Pulsed high after the INTERRUPT has been received.
DOORBELL	I	Pulsed high when PC sends a doorbell to the IP core.
DOORBELL_ERR	I	If high when DOORBELL is pulsed, indicates an error signal with the doorbell.
DOORBELL_LEN	I	If non-zero, indicates the amount of data received in the IP core's buffer (from the PC).
DOORBELL_ARG	I	Contains a "function call" style 32 bit argument of data. Asserted after DOORBELL's pulse.
DMA_REQ	O	Assert high to request a DMA transfer.
DMA_REQ_ACK	I	Pulsed high when DMA request has been received.
DMA_SRC	O	The system bus starting address of the transfer source. Valid when DMA_REQ is asserted.
DMA_DST	O	The system bus starting address of the transfer dest. (sink). Valid when DMA_REQ is asserted.
DMA_LEN	O	The length of the transfer in bytes.
DMA_SIG	O	If high, the PC on the channel will receive an interrupt after the transfer completes. Only used when DMA_REQ is asserted.
DMA_DONE	I	Pulsed high when the transfer is complete.
DMA_ERR	I	If high, indicates an error in the transfer. Only used when DMA_DONE is pulsed.
BUF_REQ	O	Assert high to request a PC buffer.
BUF_REQ_ACK	I	Pulsed high when the buffer request has been received.
BUF_REQ_ADDR	I	PC buffer system bus address.
BUF_REQ_SIZE	I	PC buffer size in powers of 2 (e.g. a value of 10 means $2^{10} = 1024$ bytes).
BUF_REQ_RDY	I	Pulsed when PC buffer information is ready and asserted.
BUF_REQ_ERR	I	If high, indicates error in receiving the PC buffer info. Used when BUF_REQ_RDY is pulsed.
BUF_REQD	I	Pulsed high when a buffer is requested from the IP core.
BUF_REQD_ADDR	O	IP core buffer system bus address.
BUF_REQD_SIZE	O	IP core buffer size in powers of 2 (e.g. a value of 10 means $2^{10} = 1024$ bytes).
BUF_REQD_RDY	O	Pulse high when the IP core buffer is ready.
BUF_REQD_ERR	O	If high, indicates error in allocating the IP core buffer. Used when BUF_REQD_RDY is pulsed.

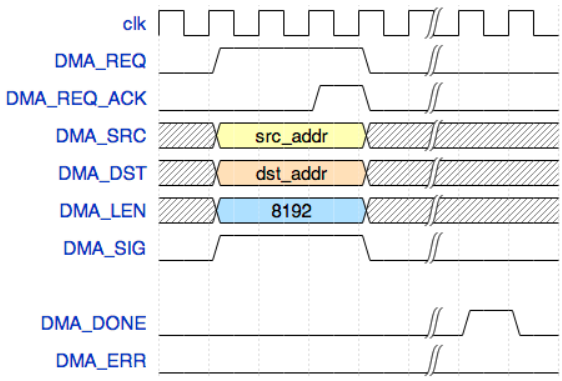


Figure A.2. RIFFA 1.0 timing diagram for DMA transfer.

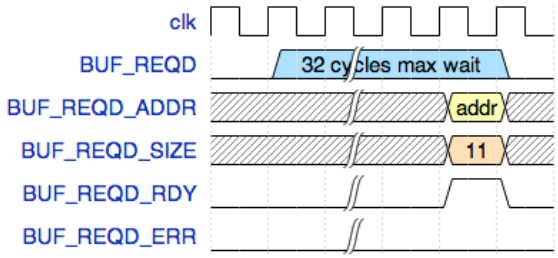


Figure A.3. RIFFA 1.0 timing diagram for FPGA buffer request.

8 KB. The DMA_SRC, DMA_DST, DMA_LEN, and DMA_SIG ports must be valid when the DMA_REQ port is asserted. The DMA_REQ port must be held high until the DMA_REQ_ACK port is pulsed. The DMA_SIG high means that after the transfer is complete, the PC will receive an interrupt containing the length of the transferred data. After the transfer is complete the DMA_DONE port is pulsed. If there was an error during the transfer, the DMA_ERR signal would be high during the DMA_DONE pulse. The DMA_ERR port is only valid when the DMA_DONE is pulsed.

Figure A.3 shows the timing for a FPGA buffer request. When the PC transfers data to the FPGA, it requests a FPGA buffer for the destination. This buffer is specified by the IP core on the channel. The IP core must provide a system bus address and a size in terms of powers of 2 (e.g. a value of 11 means $2^{11} = 2048$). Each request may be satisfied by the same buffer or different buffers. This is left up to the IP core. In the

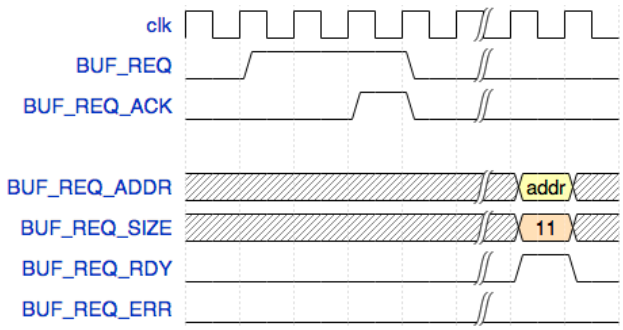


Figure A.4. RIFFA 1.0 timing diagram for PC buffer request.

example above the buffer is specified at address addr with a size of 2048 bytes. As seen earlier, the transfer of data to this buffer (from the PC) will result in a doorbell with the length of the bytes transferred (which will always be $\leq$ buffer size). Note that the BUF_REQD port will only be held high for 32 cycles. If the IP core does not provide a response and pulse BUF_REQ_RDY within 32 cycles of BUF_REQD going high, the PC will receive an error condition.

The diagram in Figure A.4 shows the timing for a PC buffer request. When the FPGA transfers data to the PC, it requests a PC buffer for the destination. This buffer is specified by the PC thread on the channel. The PC must provide a system bus address and a size in terms of powers of 2 (e.g. a value of 11 means $2^{11} = 2048$). Each request may be satisfied by the same buffer or different buffers. This is determined by the available buffers on the PC. In the example above, the PC buffer is specified at address addr with a size of 2048 bytes. Similar to the transfer of data from the PC to the FPGA, when data is received in the PC buffer, the PC will receive an interrupt with the length of the bytes transferred (which will always be $\leq$ buffer size). The BUF_REQ port must be held high until the BUF_REQ_ACK. There is no bound on the number of cycles between when a PC buffer is requested and when a response will be returned to the IP core. If the PC does not provide a response and pulse BUF_REQ_RDY within a time frame required by

Table A.2. RIFFA 1.0 SIMPBUS hardware interface.

Signal Name	I/O	Description
SIMPBUS_ADDR	O	Bus address.Should be as wide as the system bus address to which it is connected (32, 64, etc.).
SIMPBUS_WDATA	O	Data to be written to the bus.
SIMPBUS_RDATA	I	Data read from the bus.
SIMPBUS_BE	O	Byte enable for reading and writing from/to the bus.
SIMPBUS_RNW	O	Read not write flag.
SIMPBUS_START	O	Start (active high) level indicator. Must be held high until SIMPBUS_DONE is asserted.
SIMPBUS_DONE	I	Done (active high) level indicator. Will be high for 1 cycle.
SIMPBUS_ERR	I	Error (active high) level indicator. Will be high for 1 cycle. Only valid when SIMPBUS_DONE is high.

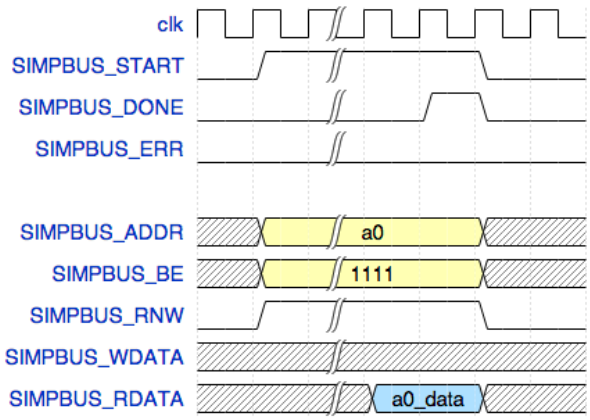


Figure A.5. RIFFA 1.0 timing diagram for SIMPBUS read.

the IP core, the IP core should issue an interrupt with an error condition.

The SIMPBUS interface is a simplified address bus protocol that allows components to be ported between specific bus protocols (e.g. Xilinx PLB, AXI, etc.). This interface is also used in the RIFFA framework. Table A.2 describes the ports. The input/output designations correspond to the master arrangement. The slave arrangement is inverted.

Timing diagrams common scenarios using the SIMPBUS interface follow.

In the diagram in Figure A.5 we see a read transaction. Note that the SIMPBUS_START port is held high until the SIMPBUS_DONE port is pulsed. If the SIMP-

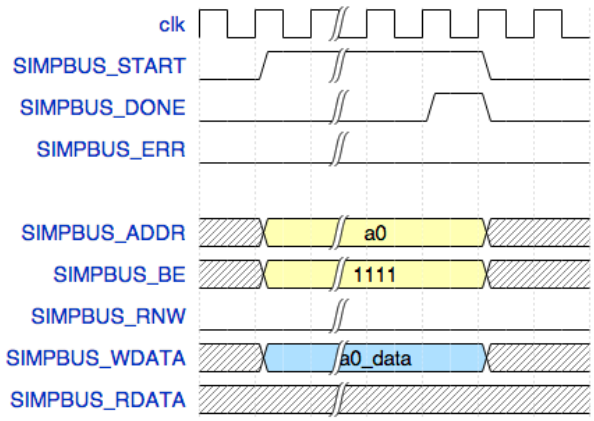


Figure A.6. RIFFA 1.0 timing diagram for SIMPBUS write.

BUS_ERR were high during the SIMPBUS_DONE pulse, this would indicate an error in reading. The SIMPBUS_ADDR, SIMPBUS_BE, and SIMPBUS_RNW should all be valid while the SIMPBUS_START is held high. The SIMPBUS_RDATA is only valid during the SIMPBUS_DONE pulse.

In the diagram in Figure A.6 we see a write transaction. Again, notice that the SIMPBUS_START port is held high until the SIMPBUS_DONE port is pulsed. If the SIMPBUS_ERR were high during the SIMPBUS_DONE pulse, this would indicate an error in writing. The SIMPBUS_ADDR, SIMPBUS_BE, SIMPBUS_RNW, and SIMPBUS_WDATA should all be valid while the SIMPBUS_START is held high.

A.3 Software API

The interface to software is a C/C++ API. This is available to user applications and works closely with the driver to provide an intuitive and flexible way to communicate with IP cores. The API is based on the notion of channels. There are 16 independent channels that can be established between user space software and the FPGA. Each channel can be driven by a single thread. The channel thread could be reading data from the FPGA, writing data to the FPGA, waiting for an interrupt or sending a doorbell. The

API is listed below:

```
int fpga_init(fpga_dev ** fpgaDev);

Initializes the FPGA memory/resources and updates the pointers in the
    fpga_dev struct. Should be called before accessing the FPGA.

fpgaDev - Handle to fpga_dev structure to initialize.

Returns:
0 on success. On error, returns:
-1 if could not open the virtual device file (check errno for details
   ).
-ENOMEM if could not map the internal buffer memory to user space.
```

```
void fpga_free(fpga_dev * fpgaDev);

Cleans up memory/resources for the FPGA virtual files. Should be
   called when done accessing the FPGA.

fpgaDev - Pointer to initialized fpga_dev structure.

Returns:
Nothing
```

```
unsigned int fpga_flip_endian(unsigned int val);

Flips an integer (32 bits) endian-ness.

val - Value to flip.

Returns:
```

Flipped unsigned int value.

```
int fpga_call_args_data(fpga_dev * fpgaDev, int channel, unsigned int
    arg0, unsigned int arg1, int argc, unsigned char * senddata, int
    sendlen, unsigned char * recvdata, int recvlen);
```

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.
arg0 – First 'function call' style argument.
arg1 – Second 'function call' style argument.
argc – Number of args to send.
senddata – Data array to send.
sendlen – Length of data in the senddata array to send.
recvdata – Data array to hold response data.
recvlen – Length of recvdata array.

Initiates a transfer of data and/or 4 byte arg values to the FPGA on channel, channel. Any args will first be written. Then any specified data will be written, possibly over several transfers. After each transfer, the IP core connected to the channel will receive a doorbell with the transfer length (in bytes). After the final transfer, the IP core will receive a zero length doorbell to signal a start. When the IP core has completed processing, any return data will be transferred from the FPGA and copied into the recvdata pointer.

Note: this call and return protocol is not enforced on the FPGA IP core. So please be sure to design the IP core state machine accordingly.

Up to argc args are sent and up to sendlen bytes from the senddata pointer are transferred to the FPGA. Any return data, up to recvlen, will be copied into the recvdata pointer. Therefore,

recvdata must accomodate at least recvlen bytes.
The endianness of sent and received data is not changed, but the endianness of sent args is flipped.

Returns:
On success, returns the total number of received bytes. The amount of bytes written to the recvdata pointer will be the minimum of the return value and recvlen. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/dropped, or some other error is encountered during transfer.
-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_call_args(fpga_dev * fpgaDev, int channel, unsigned int arg0
    , unsigned int arg1, int argc, unsigned char * recvdata, int
    recvlen);
```

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
arg0 - First 'function call' style argument.
arg1 - Second 'function call' style argument.
argc - Number of args to send.
recvdata - Data array to hold response data.
recvlen - Length of recvdata array.

Initiates a transfer of 4 byte arg values to the FPGA on channel, channel. Any args will first be written. Then the IP core will

receive a zero length doorbell to signal a start. When the IP core has completed processing, any return data will be transferred from the FPGA and copied into the recvdata pointer.

Note: this call and return protocol is not enforced on the FPGA IP core. So please be sure to design the IP core state machine accordingly.

Up to argc args are sent. Any return data, up to recvlen, will be copied into the recvdata pointer. Therefore, recvdata must accomodate at least recvlen bytes.

The endianness of received data is not changed, but the endianness of sent args is flipped.

Returns:

On success, returns the total number of received bytes. The amount of bytes written to the recvdata pointer will be the minimum of the return value and recvlen. On error, returns:

-EACCES if the channel is not open.

-ETIMEDOUT if timeout is non-zero and expires before all data is received.

-EREMOTEIO if the transfer sequence takes too long, data is lost/dropped, or some other error is encountered during transfer.

-ERESTARTSYS if a signal interrupts the thread.

-ENOMEM if the driver runs out of buffers for data transfers.

-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_call_data(fpga_dev * fpgaDev, int channel, unsigned char *
    senddata, int sendlen, unsigned char * recvdata, int recvlen);
```

fpgaDev – Pointer to initialized fpga_dev structure.

channel – Channel number over which to communicate.

senddata – Data array to send.

sendlen – Length of data in the senddata array to send.

recvdata – Data array to hold response data.

recvlen – Length of recvdata array.

Initiates a transfer of data to the FPGA on channel, channel. Any specified data will be written, possibly over several transfers. After each transfer, the IP core connected to the channel will receive a doorbell with the transfer length (in bytes). After the final transfer, the IP core will receive a zero length doorbell to signal a start. When the IP core has completed processing, any return data will be transferred from the FPGA and copied into the recvdata pointer.

Note: this call and return protocol is not enforced on the FPGA IP core. So please be sure to design the IP core state machine accordingly.

Up to sendlen bytes from the senddata pointer are transferred to the FPGA. Any return data, up to recvlen, will be copied into the recvdata pointer. Therefore, recvdata must accomodate at least recvlen bytes.

The endianness of sent and received data is not changed.

Returns:

On success, returns the total number of received bytes. The amount of bytes written to the recvdata pointer will be the minimum of the return value and recvlen. On error, returns:

-EACCES if the channel is not open.

-ETIMEDOUT if timeout is non-zero and expires before all data is received.

-EREMOTEIO if the transfer sequence takes too long, data is lost/dropped, or some other error is encountered during transfer.

-ERESTARTSYS if a signal interrupts the thread.

-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_call(fpga_dev * fpgaDev, int channel, unsigned char *
    recvdata, int recvlen);
```

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.
recvdata – Data array to hold response data.
recvlen – Length of recvdata array.

Sends a zero length doorbell to signal a start. When the IP core has completed processing, any return data will be transferred from the FPGA and copied into the recvdata pointer.
Note: this call and return protocol is not enforced on the FPGA IP core. So please be sure to design the IP core state machine accordingly.
Any return data, up to recvlen, will be copied into the recvdata pointer. Therefore, recvdata must accomodate at least recvlen bytes.
The endianness of received data is not changed.

Returns:
On success, returns the total number of received bytes. The amount of bytes written to the recvdata pointer will be the minimum of the return value and recvlen. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/dropped, or some other error is encountered during transfer.

-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_send_args_data(fpga_dev * fpgaDev, int channel, unsigned int
    arg0, unsigned int arg1, int argc, unsigned char * senddata, int
    sendlen, int start);
```

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.
arg0 – First 'function call' style argument.
arg1 – Second 'function call' style argument.
argc – Number of args to send.
senddata – Data array to send.
sendlen – Length of data in the senddata array to send.
start – If 1, a zero length doorbell will signal a start after the transfer of data.

Writes 4 byte arg values and/or data to the FPGA on channel, channel. Up to argc args will be written. After all the args have been written, sendlen bytes from the senddata pointer will be written, possibly over multiple transfers. After each transfer, the IP core connected to the channel will receive a doorbell with the transfer length (in bytes). If start == 1, then after the final transfer, the IP core will receive a zero length doorbell to signal start.
The endianness of sent data is not changed, but the endianness of sent args is flipped.

Returns:
0 on success. On error, returns:

-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/dropped, or some other error is encountered during transfer.
-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_send_args(fpga_dev * fpgaDev, int channel, unsigned int arg0
    , unsigned int arg1, int argc, int start);
```

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
arg0 - First 'function call' style argument.
arg1 - Second 'function call' style argument.
argc - Number of args to send.
start - If 1, a zero length doorbell will signal a start after the transfer of args.

Writes 4 byte arg values to the FPGA on channel, channel. Up to argc args will be written. After all the args have been written, if start == 1, the IP core connected to the channel will receive a zero length doorbell to signal start.
The endianness of sent args is flipped.

Returns:
0 on success. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is received.

-EREMOTEIO if the transfer sequence takes too long, data is lost/ dropped, or some other error is encountered during transfer.
-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

---

int fpga_send_data(fpga_dev * fpgaDev, int channel, unsigned char * senddata, int sendlen, int start);

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.
senddata – Data array to send.
sendlen – Length of data in the senddata array to send.
start – If 1, a zero length doorbell will signal a start after the transfer of data.

Writes data to the FPGA on channel, channel. All sendlen bytes from the senddata pointer will be written (possibly over multiple transfers). After each transfer, the IP core connected to the channel will receive a doorbell with the transfer length (in bytes). If start == 1, then after the final transfer, the IP core will receive a zero length doorbell to signal start.
The endianness of sent data is not changed.

Returns:
0 on success. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/ dropped, or some other error is encountered during transfer.

-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_send_doorbell(fpga_dev * fpgaDev, int channel, int err);
```

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
err - If 1, the doorbell will contain an error signal.

Sends a zero length doorbell to the IP core connected to the channel.
    If err == 1, an error will be signaled along with the doorbell.

Returns:
0 on success. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is
   received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/
   dropped, or some other error is encountered during transfer.
-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_send_data_begin(fpga_dev * fpgaDev, int channel, unsigned
   char * senddata, int sendlen, int start);
```

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
senddata - Data array to send.
sendlen - Length of data in the senddata array to send.

start - If 1, a zero length doorbell will signal a start after the transfer of data.

Spawns an internal thread to send data to the FPGA on channel, channel, from the senddata pointer. Up to sendlen bytes will be copied from the senddata pointer, possibly over multiple transfers. After each transfer, the IP core connected to the channel will receive a doorbell with the transfer length (in bytes). If start == 1, then after the final transfer, the IP core will receive a zero length doorbell to signal start. The function fpga_send_data_end can be used to wait for completion using the
same channel. This function will return immediately.

Returns:
0 on success. On error, returns:
-EACCES if the channel is not open.
-EAGAIN if an internal thread cannot be created.

---

int fpga_send_data_end(fpga_dev * fpgaDev, int channel);

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.

Waits for an internal thread created by the fpga_send_data_begin function to send all data to the FPGA on channel, channel. This function will block until all data on the channel is sent.
The endianness of received data is not changed.

Returns:
0 on success. On error, returns:

-EACCES if the channel is not open.
-EINVAL if the thread is not joinable.
-ESRCH if no send thread has been created for the channel.
-EDEADLK if deadlock was detected.

```
int fpga_recv_data(fpga_dev * fpgaDev, int channel, unsigned char *
    recvdata, int recvlen);
```

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
recvdata - Data array to hold response data.
recvlen - Length of recvdata array.

Reads data from the FPGA on channel, channel, to the recvdata pointer . Up to recvlen bytes will be copied to the recvdata pointer (possibly over multiple transfers). Therefore, recvdata must accomodate at least recvlen bytes.
The endianness of received data is not changed.

Returns:
The number of bytes received on the channel. The number of bytes written to the recvdata pointer will be the minimum of return value and recvlen. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/ dropped, or some other error is encountered during transfer.
-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_wait_interrupt(fpga_dev * fpgaDev, int channel);

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.

Waits for an interrupt to be received on the channel. Equivalent to
   waiting for a zero length receive data interrupt.

Returns:
0 on success. On error, returns:
-EACCES if the channel is not open.
-ETIMEDOUT if timeout is non-zero and expires before all data is
   received.
-EREMOTEIO if the transfer sequence takes too long, data is lost/
   dropped, or some other error is encountered during transfer.
-ERESTARTSYS if a signal interrupts the thread.
-ENOMEM if the driver runs out of buffers for data transfers.
-EFAULT if internal queues are exhausted or on bad address access.
```

```
int fpga_recv_data_begin(fpga_dev * fpgaDev, int channel, unsigned
   char * recvdata, int recvlen);

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
recvdata - Data array to hold response data.
recvlen - Length of recvdata array.

Spawns an internal thread to receive data from the FPGA on channel,
   channel, to the recvdata pointer. Up to recvlen bytes will be
   copied to the recvdata pointer, possibly over multiple transfers.
```

Therefore, recvdata must accommodate at least recvlen bytes. The function fpga_recv_data_end can be used to wait for completion using the same channel. This function will return immediately.

Returns:
0 on success. On error, returns:
-EACCES if the channel is not open.
-EAGAIN if an internal thread cannot be created.

---

int fpga_recv_data_end(fpga_dev * fpgaDev, int channel);

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.

Waits for an internal thread created by the fpga_recv_data_begin function to receive all data from the FPGA on channel, channel. This function will block until all data on the channel is received.
The endianness of received data is not changed.

Returns:
On success, the number of bytes actually received on the channel are returned. The number of bytes written to the original recvdata pointer will be the minimum of the returned value and the original recvlen value (see the fpga_recv_data_begin function). On error, returns:
-EACCES if the channel is not open.
-EINVAL if the thread is not joinable.
-ESRCH if no receive thread has been created for the channel.
-EDEADLK if deadlock was detected.

```
int fpga_config_read(fpga_dev * fpgaDev, int offset, unsigned int *
   val);

fpgaDev – Pointer to initialized fpga_dev structure.
offset – Offset from the start of the configuration address space.
val – Pointer into which to copy the configuration value.

Reads the FPGA config space value specified at offset, offset, into
   the val pointer. The value of offset specifies the number of
   bytes from the start of the config address space. The offset
   value must be word aligned. Note: this function accesses the FPGA
    configuration address space, so you should know what
you're doing when using this.
The endianness of read data is flipped.

Returns:
0 on success. On error, returns:
-EFAULT if wordoffset is not within the config space.
```

```
int fpga_config_write(fpga_dev * fpgaDev, int offset, unsigned int
   val);

fpgaDev – Pointer to initialized fpga_dev structure.
offset – Offset from the start of the configuration address space.
val – Configuration value to write.

Writes the value of val to the FPGA config space at offset, offset.
   The value of offset specifies the number of bytes from the start
   of the config address space. The offset value must be word
   aligned. Note: this function accesses the FPGA configuration
   address space, so you should know what you're doing when using
```

this.
The endianness of written data is flipped.

Returns:
0 on success. On error, returns:
-EFAULT if wordoffset is not within the config space.

---

int fpga_mem_copy(fpga_dev * fpgaDev, int channel, unsigned int srcaddr, unsigned int dstaddr, unsigned int len, int doorbell, int wait);

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.
srcaddr – Source start address from which data will be copied.
dstaddr – Destination start address to which data will be copied.
len – Length of data to copy, in bytes.
doorbell – If 1, sends a doorbell to the connected IP core on this channel with the transfer length, after the transfer is complete.
wait – If 1, waits until the transfer is complete before returning.

Requests a transfer of data from srcaddr to dstaddr on the FPGA of length len, using channel, channel. All len bytes will be copied in a single transfer. After the transfer, if doorbell == 1, the IP core connected to the channel will receive a doorbell with the transfer length (in bytes). If wait == 1, this function will wait until the transfer has completed before returning. Otherwise, this function will return immediately after initiating the transfer. Note: this function takes FPGA bus addresses and thus assumes you know what you're doing.
The endianness of copied data is not changed.

Returns:

0 on success. On error, returns:

-EACCES if the channel is not open.

-ETIMEDOUT if timeout is non-zero and expires before all data is received.

-EREMOTEIO if the transfer sequence takes too long, data is lost/dropped, or some other error is encountered during transfer.

-ERESTARTSYS if a signal interrupts the thread.

-ENOMEM if the driver runs out of buffers for data transfers.

-EFAULT if internal queues are exhausted or on bad address access.

```
int fpga_remote_buf_allocate(fpga_dev * fpgaDev, int channel, int *
    size, unsigned int * addr);
```

fpgaDev – Pointer to initialized fpga_dev structure.

channel – Channel number over which to communicate.

size – Pointer to buffer size.

addr – Pointer to buffer system bus address.

Requests a FPGA buffer be allocated. The address size and addr will be specified on success. The addr will be a FPGA system bus address and can be used with the fpga_mem_copy function.

Returns:

0 on success. On error, returns a non-zero value.

```
int fpga_internal_buf_allocate(fpga_dev * fpgaDev, int channel, int *
     size, unsigned int * addr, unsigned char ** buf, int * bar, int
    * segment);
```

fpgaDev – Pointer to initialized fpga_dev structure.

channel – Channel number over which to communicate.
size – Pointer to buffer size.
addr – Pointer to buffer system bus address.
buf – Handle to buffer location.
bar – Internal memory allocation identifier.
segment – Internal memory allocation identifier.

Requests an internal buffer be allocated. The addr, size, bar, buf, and segment will be specified on success. The bar and segment are needed to free the buffer after use. The addr is a FPGA bus address and can be used with the fpga_mem_copy function. The buf points to the buffer in user space and can be accessed directly.

Returns:
0 on success. On error, returns a non-zero value.

---

int fpga_internal_buf_free(fpga_dev * fpgaDev, int channel, int bar, int segment);

fpgaDev – Pointer to initialized fpga_dev structure.
channel – Channel number over which to communicate.
bar – Internal memory allocation identifier.
segment – Internal memory allocation identifier.

Frees the internal buffer so that it can be reused. The bar and segment values are returned from the fpga_buf_allocate function and specify the buffer.

Returns:
0 on success. On error, returns a non-zero value.

```
int fpga_channel_open(fpga_dev * fpgaDev, int channel, int timeout);

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.
timeout - Fail safe timeout for all function calls (so program does
   not freeze waiting on FPGA).

Opens the specified channel. Valid channels are in the range [0,15].
   The timeout value sets a threshold in ms for blocking for all
   channel operations. It is meant to avoid infinite blocking in
   case of errors. When the timeout is exceeded, the operation
   returns a failure code. Timed-out functions will not reliably
   return partial results or execute partial sends of any data. A
   value of 0 indicates an indefinite wait (no timeout). Only non-
   negative values are valid.

Returns:
0 on success. On error, returns:
-1 if could not open the virtual file (check errno for details).
```

```
void fpga_channel_close(fpga_dev * fpgaDev, int channel);

fpgaDev - Pointer to initialized fpga_dev structure.
channel - Channel number over which to communicate.

Closes the specified channel and releases any internal resources.

Returns:
Nothing.
```

```
int fpga_channel_timeout(fpga_dev * fpgaDev, int channel, int timeout
    );
```

fpgaDev – Pointer to initialized fpga_dev structure.

channel – Channel number over which to communicate.

timeout – Fail safe timeout for all function calls (so program does not freeze waiting on FPGA).

Sets the channel timeout to the new value. The timeout value sets a threshold in ms for blocking for all channel operations. It is meant to avoid infinite blocking in case of errors. When the timeout is exceeded, the operation returns a failure code. Timed-out functions will not reliably return partial results or execute partial sends of any data. A value of 0 indicates an indefinite wait (no timeout). Only non-negative values are valid.

Returns:

0 on success. On error, returns a non-zero value.

Appendix B

RIFFA 2.0

B.1 Getting Started

The RIFFA 2.0 distribution contains the RIFFA source HDL, the RIFFA driver and software bindings, an installation script, and examples for common FPGA development boards. This website and the distribution should help you get started if you're using one of the FPGA boards we've tested. If you're using a different board, you'll need to adapt the setup instructions accordingly.

There are only really 2 steps to get a basic RIFFA 2.0 design up and running.

Step 1: Install RIFFA on your system

Linux: Download the RIFFA 2.0 distribution. Build and install the kernel driver and C/C++ library as:

```
sudo make setup
make
sudo make install
```

You'll only need to run `make setup` once, as this simply attempts to install the Linux kernel headers for your version of the Linux kernel. They won't need updating

unless you upgrade your system kernel. Running `make` (or `make debug`) will build the driver and C/C++ library. Using the `make debug` directive will output debug messages in the kernel log. This can be very helpful when you're developing your application. After running `make install`, you'll want to reboot to let the system find your RIFFA 2.0 design on the PCIe bus and let the OS load your driver.

You can install the Java and Python bindings by following the directions for those binding in their respective directories within the RIFFA 2.0 distribution.

Windows: Download the RIFFA 2.0 distribution. Install the kernel driver and C/C++ library by running the setup.exe or setup_dbg.exe installer (the debug installer outputs additional debug messages to the Windows debug framework). After running the installer, reboot to let the system find your RIFFA 2.0 design on the PCIe bus and let the OS load your driver.

You can install the Java and Python bindings by following the directions for those binding in their respective directories within the RIFFA 2.0 distribution.

Step 2: Create and build a RIFFA design for your FPGA. As described on the architecture page, RIFFA 2.0 relies on a PCIe Endpoint core to drive the transceivers. We have produced step by step guides to build a design:

- Design Guide - Avnet Xilinx S6LX150t - ISE in Appendix B.7
- Design Guide - Xilinx ML605 - ISE in Appendix B.8
- Design Guide - Xilinx VC707 - ISE in Appendix B.9
- Design Guide - Xilinx VC707 - Vivado in Appendix B.10

If you have a development board that is not covered by one of our guides, let us know. We would be happy to test with a PCIe Endpoint core for that board and add it to

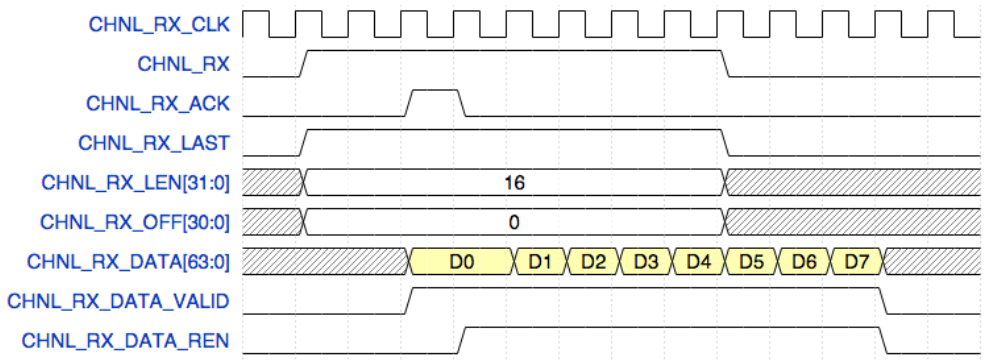


Figure B.1. RIFFA 2.0 timing diagram for receiving.

the list. However, we have decided not to support older FPGAs with legacy interfaces, such as the Xilinx Virtex 5. If you need Virtex 5 support consider using RIFFA 1.0.

B.2 Hardware Interface

The RIFFA 2.0 channel has two sets of signals, one for receiving data (RX) and one for sending data (TX). This version has simplified the interface to use a minimal handshake and receive/send data using a FIFO with first word fall through semantics (valid+read interface). The clocks used for receiving and sending can be asynchronous from each other and from the PCIe interface (RIFFA clock). Table B.1 describes the ports. The input/output designations are from your user core's perspective (i.e. the core(s) you write and connect to the RIFFA 2.0 channel).

Figure B.2 is a timing diagram for receiving data. The timing diagram shows the RIFFA channel receiving a data transfer of 16 (4 byte) words (64 bytes). When `CHNL_RX` is high, `CHNL_RX_LAST`, `CHNL_RX_LEN`, and `CHNL_RX_OFF` will all be valid. In this example, `CHNL_RX_LAST` is high, indicating to the user core that there are no other transactions following this one and that the user core can start processing the received data as soon as the transaction completes. `CHNL_RX_LAST` may be set low if multiple transactions will be initiated before the user core should start processing

Table B.1. RIFFA 2.0 hardware interface. The value of DWIDTH will be either 32, 64, or 128.

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.

received data. Of course, the user core will always need to read the data as it arrives, even if CHNL_RX_LAST is low.

In the example CHNL_RX_OFF is 0. However, if the PC specified a value for offset when it initiated the send, that value would be present on the CHNL_RX_OFF signal. The 31 least significant bits of the 32 bit integer specified by the PC thread are transmitted (due to packing constraints). The CHNL_RX_OFF signal is meant to be used in situations where data is transferred in multiple sends and the user core needs to know where to write the data (if, for example it is writing to BRAM or DRAM).

The user core must pulse the CHNL_RX_ACK signal high for at least one cycle to acknowledge the receive transaction. The RIFFA channel will not recognize that the transaction has been received until it receives a CHNL_RX_ACK pulse. Note that data on CHNL_RX_DATA may begin to arrive before CHNL_RX_ACK is pulsed, but the FIFO will never overflow. The combination of CHNL_RX_DATA_VALID high and CHNL_RX_DATA_REN high consumes the data on CHNL_RX_DATA. New data will be provided until the FIFO is drained. Note that the FIFO may drain completely before all the data has been received. The CHNL_RX signal will remain high until all data for the transaction has been received into the FIFO. Note that CHNL_RX may go low while CHNL_RX_DATA_VALID is still high. That means there is still data in the FIFO to be read by the user core. Attempting to read (asserting CHNL_RX_DATA_REN high) while CHNL_RX_DATA_VALID is low, will have no affect on the FIFO. The user core may want to count the number of words received and compare against the value provided by CHNL_RX_LEN to keep track of how much data is expected.

In the event of a transmission error, the amount of data received may be less than the amount expected (advertised on CHNL_RX_LEN). It is the user core's responsibility to detect this discrepancy if important to the user core.

The diagram in Figure B.2 shows the RIFFA channel sending a data transfer of

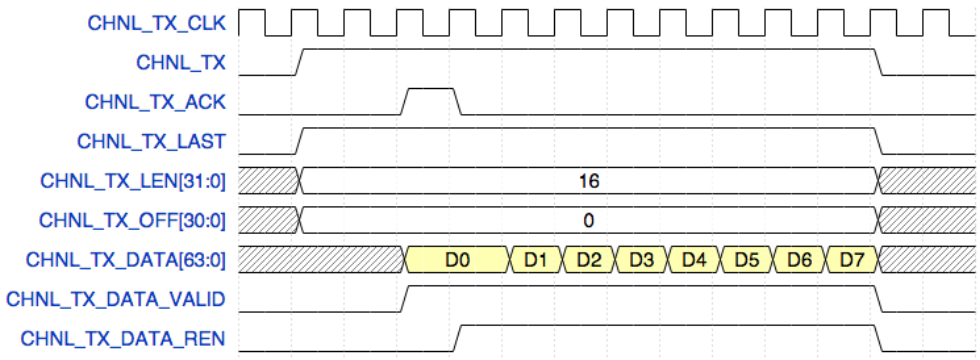


Figure B.2. RIFFA 2.0 timing diagram for sending.

16 (4 byte) words (64 bytes). It's nearly symmetric to the receive example. The user core sets `CHNL_TX` high and asserts values for `CHNL_TX_LAST`, `CHNL_TX_LEN`, and `CHNL_TX_OFF` for the duration `CHNL_TX` is high. `CHNL_TX` must remain high until all data has been consumed. RIFFA will expect to read `CHNL_TX_LEN` words from the user core. Any more data provided may be consumed, but will be discarded. The user core can provide less than `CHNL_TX_LEN` words and drop `CHNL_TX` at any point. Dropping `CHNL_TX` indicates the end of the transaction. Whatever data was consumed before `CHNL_TX` was dropped will be sent and reported as received to the software thread.

As with the receive interface, setting `CHNL_TX_LAST` high will signal to the PC thread to not wait for additional transactions (after this one). Setting `CHNL_TX_OFF` will cause the transferred data to be written into the PC thread's buffer starting `CHNL_TX_OFF` 4 bytes words from the beginning. This can be useful when sending multiple transactions and needing to order them in the PC thread's receive buffer. `CHNL_TX_LEN` defines the length of the transaction in 4 byte words.

As the `CHNL_TX_DATA` bus can be 32 bits, 64 bits, or 128 bits wide, it may be that the number of 32 bit words the user core wants to transfer is not an even multiple of the bus width. In this case, `CHNL_TX_DATA_VALID` must be high on the last cycle `CHNL_TX_DATA` has at least 1 word to send. The channel will only send as many words

as is specified by CHNL_TX_LEN. So any additional data consumed, past the last word, will be discarded.

After CHNL_TX goes high, the RIFFA channel will pulse high the CHNL_TX_ACK and begin to consume data on the CHNL_TX_DATA bus. When CHNL_TX_DATA_VALID is high and CHNL_TX_DATA_REN is high, data on CHNL_TX_DATA will be consumed. New data can be consumed every cycle. After all the data is consumed, CHNL_TX can be dropped. Keeping CHNL_TX_DATA_VALID high while CHNL_TX_DATA_REN is low will have no effect.

B.3 C/C++ API

The software interface is provided by bindings for C/C++, Python, and Java. After installation all bindings are available in their respective runtime environments. The API is based on the notion of channels. RIFFA 2.0 can be configured to support between 1 - 12 independent channels. Each channel connects to an IP core and can be addressed by specifying the channel number from the user application. The channels are independent and thread safe. At most one thread should be used to access a single channel.

The C/C++ bindings are used by including the <riffa.h> header file and linking with the -lriffa library. Belowis a complete example and an API listing.

```
#include <stdio.h>
#include <stdlib.h>
#include <riffa.h>

#define BUF_SIZE (1*1024*1024)
unsigned int buf[BUF_SIZE];

int main(int argc, char* argv[]) {
  fpga_t * fpga;
```

```
    int fid = 0; // FPGA id
    int channel = 0; // FPGA channel

    fpga = fpga_open(fid);
    fpga_send(fpga, channel, (void *)buf, BUF_SIZE, 0, 1, 0);
    fpga_recv(fpga, channel, (void *)buf, BUF_SIZE, 0);
    fpga_close(fpga);
    return 0;
}
```

This example first opens up FPGA with id 0. It then sends 4 MB of data (1 mega-words) to channel 0 with 0 destination offset, 0 timeout, and marks the transfer as last. The IP core on channel 0 is designed to send back some data. So the next call is to receive data from the same channel, up to 4 MB, with 0 timeout. Note a few things:

- We're using the same buffer to send data and receive data. This is not required, it just makes for a simpler example.
- The timeout is set to 0, which may hang the application if there's a problem with the IP core logic.
- In practice, you'd want to check the return values to see how much data was sent and received. You'd also probably want some error handling.

Using gcc, this example can be compiled as:

```
gcc tester.c -lriffa
```

Using Microsoft Visual Studio, you'll need to set your project configuration properties to find the `riffa.h` header file and add the import library as a dependency. Under C/C++ → General → Additional Include Directories, specify the path to the `riffa.h` header file under. Under Linker → Input → Additional Dependencies, specify the path

to `riffa.lib`. Visual Studio should be able to find the `riffa.lib` without adding an additional dependency path.

B.3.1 API

The API is provided in the listing below.

```
int fpga_list(fpga_info_list * list);

Populates the fpga_info_list pointer with all FPGAs registered in the
    system. See riffa_driver.h for the fpga_info_list definition.
   Returns 0 on success, a negative value on error.

list - Pointer to a fpga_info_list struct to populate.

Returns:
0 on success, a negative value on error.
```

```
fpga_t * fpga_open(int id);

Initializes the FPGA specified by id. On success, returns a pointer
   to a fpga_t struct. On error, returns NULL. Each FPGA must be
   opened before any channels can be accessed. Once opened, any
   number of threads can use the fpga_t struct pointer.

id - Identifier for the FPGA (in single FPGA installations, this is
   always 0).

Returns:
A fpga_t struct pointer or NULL.
```

```
void fpga_close(fpga_t * fpga);
```

Cleans up memory/resources for the FPGA specified by the descriptor (pointer).

fpga – Pointer to fpga_t struct.

Returns:
Nothing.

---

int fpga_send(fpga_t * fpga, int chnl, void * data, int len, int destoff, int last, long long timeout);

fpga – Pointer to fpga_t structure.
chnl – Channel number over which to communicate.
data – Pointer to array of data to send. Note that the data transfer unit is a 32 bit word.
len – Length of data to send, in (32 bit) words. Thus a value of 4 means send 16 bytes.
destoff – Value sent to FPGA core to indicate where to start writing this data. Only the least significant 31 bits are sent (not all 32).
last – If 1, this transfer is the last in a sequence of transfers. If 0, this transfer is not the last in a sequence of transfers (more transfers to come).
timeout – Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

Sends len words (4 byte words) from data to FPGA channel chnl using the fpga_t struct. The FPGA channel will be sent len, destoff, and last. The value of destoff is used to support sending data

across multiple send transactions. Note that only the low 31 bits of this unsigned int are sent. If last is 1, the channel should interpret the end of this send as the end of a transaction. If last is 0, the channel should wait for additional sends before the end of the transaction. If timeout is non-zero, this call will send data and wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero, this call may block indefinitely. Multiple threads sending on the same channel may result in corrupt data or error. This function is thread safe across channels. Returns the number of words sent.

Returns:
The number of words sent.

---

int fpga_recv(fpga_t * fpga, int chnl, void * data, int len, long long timeout);

fpga - Pointer to fpga_t structure.
chnl - Channel number over which to communicate.
data - Pointer to buffer array where received data will be written.
len - Length of buffer array, in (32 bit) words. Thus a value of 4 means send 16 bytes.
timeout - Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

Receives data from the FPGA channel chnl to the data pointer, using the fpga_t struct. The FPGA channel can send any amount of data, so the data array should be large enough to accommodate. The len parameter specifies the actual size of the data buffer in words (4 byte words). The FPGA will specify an offset value which will

```
    determine where received data will start being written. If the
    amount of data plus the offset exceed the size of the data array,
     then the additional data will be discarded. If timeout is non-
    zero, this call will wait up to timeout ms for the FPGA to
    respond (between packets) before timing out. If timeout is zero,
    this call may block indefinitely. Multiple threads receiving on
    the same channel may result in corrupt data or error. This
    function is thread safe across channels. Returns the number of
    words received to the data array.

Returns:
The number of words received to the data array.
```

```
void fpga_reset(fpga_t * fpga);

Resets the state of the FPGA and all transfers across all channels.
    This is meant to be used as an alternative to rebooting if an
    error occurs while sending/receiving. Calling this function while
     other threads are sending or receiving will result in unexpected
     behavior.

fpga - Pointer to fpga_t structure.

Returns:
Nothing.
```

B.4 Java API

The software interface is provided by bindings for C/C++, Python, and Java. After installation all bindings are available in their respective runtime environments. The API is based on the notion of channels. RIFFA 2.0 can be configured to support between 1 -

12 independent channels. Each channel connects to an IP core and can be addressed by specifying the channel number from the user application. The channels are independent and thread safe. At most one thread should be used to access a single channel.

The Java bindings are used by including the `riffa.jar` file in the classpath for compiling and running. You will need Java 1.4 or greater. Below is a complete example and an API listing.

```
import edu.ucsd.cs.riffa.*;
import java.nio.ByteBuffer;

public class MyApp {
  private static final int BUF_SIZE = 1*1024*1024;

  public void main(String[] args) throws Exception {
    int fid = 0;
    int channel = 0;

    ByteBuffer buf = ByteBuffer.allocateDirect(BUF_SIZE);
    Fpga fpga = Fpga.open(fid);
    fpga.send(channel, buf, BUF_SIZE, 0, true, 0L);
    fpga.recv(channel, buf, BUF_SIZE, 0L);
    fpga.close();
  }
}
```

This example first opens up the FPGA with id 0. It then sends 4 MB of data (1 mega-words) to channel 0 with 0 destination offset, 0 timeout, and marks the transfer as last. The IP core on channel 0 is designed to send back some data. So the next call is to receive data from the same channel, up to 4 MB, with 0 timeout. Note a few things:

- We're using a java.nio.ByteBuffer, not a byte[]. ByteBuffer's expose the underlying

byte[], which makes it easy for reading/writing. They also have methods to return different java.nio.Buffer subclasses for different primitives (e.g. java.nio.IntBuffer for int data). These methods do not copy the underlying data, they simply reinterpret it as an array of the specified primitive type. The different java.nio.Buffer subclasses provide methods to access the underlying data as an array of primitives (e.g. int[]). Avoiding unnecessary copying is crucial for maintaining high performance.

- We're using the same buffer to send data and receive data. This is not required, it just makes for a simpler example.
- The timeout is set to 0. If there's a problem with the IP core logic, a 0 timeout will cause the program to wait forever.
- In practice, you'd want to check the return values to see how much data was sent and received. You'd also probably want some error handling.

One more important note. Java uses network byte order, which is big endian. Most workstations are little endian (i.e. Intel or AMD). VHDL and Verilog are also little endian in that the left most bit is always the most significant. The ByteBuffer classes will encode/decode numeric values in big endian format. This encoding only applies to data in the ByteBuffer. Java primitives are handled correctly without any need for byte swapping. For example:

```
ByteBuffer buf = ByteBuffer.allocateDirect(4);
IntBuffer ibuf = buf.asIntBuffer();
ibuf.put(0, 4);
int v = 4;
System.out.printf("%d - 0x%02x%02x%02x%02x\n", ibuf.get(0),
    buf.get(3), buf.get(2), buf.get(1), buf.get(0));
```

```
System.out.printf("%d - 0x%02x%02x%02x%02x\n", v,
    v&0xFF000000, v&0xFF0000, v&0xFF00, v&0xFF);
```

Prints the following:

```
4 - 0x04000000
4 - 0x00000004
```

The first output line is from the ByteBuffer and shows the big endian encoding. The second is from the int and shows the little endian format. You'll only need to consider byte swapping when sending ByteBuffer payload data between the FPGA and the Java program. Byte swapping is most effectively done in hardware on the send and receive data ports.

The above example can be compiled as:

```
javac -cp riffa.jar MyApp.java
```

and run as:

```
java -cp riffa.jar:./ MyApp
```

B.4.1 API

The API is provided in the listing below.

```
edu.ucsd.cs.riffa

public class FpgaInfo
Value object to hold information about all the installed FPGA
   accessible by RIFFA.
```

```
public int getNumFpgas()

Returns the number of RIFFA accessible FPGAs installed in the system.
```

Returns:

Number of RIFFA accessible FPGAs installed in the system.

---

public int getId(int pos)

Returns the FPGA id at position pos. This id is used to open the FPGA on the Fpga's open method.

Returns:

FPGA id at position pos.

---

public int getNumChannels(int pos)

Returns the number of RIFFA channels configured on the FPGA at position pos.

Returns:

Number of RIFFA channels configured on the FPGA at position pos.

---

public String getName(int pos)

Returns the name of the FPGA at position pos. This is typically the PCIe bus and slot number.

Returns:

Name of the FPGA at position pos.

---

public int getVendorId(int pos)

Returns the FPGA vendor id at position pos.

Returns:

The FPGA vendor id at position pos.

---

public int getDeviceId(int pos)

Returns the FPGA device id at position pos.

Returns:

The FPGA device id at position pos.

---

public class Fpga

Represents a FPGA accessible by RIFFA. The usage pattern is:

```
Fpga f = Fpga.open(...);
f.send(...);
f.recv(...);
f.close(...);
```

The static method list can be used to get a listing of the FPGAs installed in the system and their ids. You'll need the FPGA id to pass to the open method. If only 1 FPGA is installed in the system, it's id will always be 0.

In the send and recv methods below use java.nio.ByteBuffer instead of a byte[] to represent the data for sending and buffer for receiving data. The java.nio.ByteBuffer class is used because the underlying byte[] can easily be accessed. But more importantly, it has methods to reinterpret the underlying byte[] into other primitive array types (e.g. int[]) without copying the contents of the array. Copying arrays (especially large arrays) will reduce throughput considerably and is best to be avoided. Use the

allocateDirect method on ByteBuffer to create a new instance. Then use one of the asXXXBufer methods to acquire the appropriate java.nio.Buffer subclass. The example below illustrates this:

```
ByteBuffer bb = ByteBuffer.allocateDirect(NUM_INTS*4);
IntBuffer ib = bb.asIntBuffer();
for (i = 0; i < NUM_INTS; i++)
    ib.put(i, ...)
Fpga f = Fpga.open(0);
fpga.send(0, bb, NUM_INTS, 0, true, 0L);
```

---

public static FpgaInfo list()

Populates and returns a FpgaInfo object with all FPGAs registered in the system. Returns a FpgaInfo object on success. Returns null on error.

Returns:
A FpgaInfo object on success or null.

---

public static Fpga open(int id)

Initializes the FPGA specified by id. On success, returns a Fpga object. On error, returns null. Each FPGA must be opened before any channels can be accessed. Once opened, any number of threads can use the Fpga object.

id - Identifier for the FPGA (in single FPGA installations, this is always 0).

Returns:

A Fpga object or null.

---

public void close()

Cleans up memory/resources for the FPGA represented by this instance.

Returns:
Nothing.

---

public int send(int chnl, java.nio.ByteBuffer data, int len, int destoff, boolean last, long timeout)

chnl – Channel number over which to communicate.
data – java.nio.ByteBuffer holding the byte[] to send. Note that the data transfer unit is a 32 bit word.
len – Length of data to send, in (32 bit) words. Thus a value of 4 means send 16 bytes.
destoff – Value sent to FPGA core to indicate where to start writing this data. Only the least significant 31 bits are sent (not all 32).
last – If true, this transfer is the last in a sequence of transfers. If false, this transfer is not the last in a sequence of transfers (more transfers to come).
timeout – Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

Sends len words (4 byte words) from data to FPGA channel chnl on the FPGA represented by this Fpga object. The FPGA channel will be sent len, destoff, and last. The java.nio.ByteBuffer's position and limit are not read (or modified). To send a subset of data

from the buffer, set the position and use the slice method to acquire a java.nio.ByteBuffer object that starts at the correct location. The value of destoff is used to support sending data across multiple send transactions. Note that only the low 31 bits of this value are sent. If last is true the channel should interpret the end of this send as the end of a transaction. If last is false, the channel should wait for additional sends before the end of the transaction. If timeout is non-zero, this call will send data and wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero, this call may block indefinitely. Multiple threads sending on the same channel may result in corrupt data or error. This function is thread safe across channels. Returns the number of words sent.

Returns:
The number of words sent.

---

public int recv(int chnl, java.nio.ByteBuffer data, long timeout)

chnl - Channel number over which to communicate.
data - java.nio.ByteBuffer into which received data will be written.
timeout - Timeout value in ms. If 0, no timeout is specified.
Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

Receives data from the FPGA channel chnl to the java.nio.ByteBuffer object, on the FPGA represented by this Fpga object. The FPGA channel can send any amount of data, so the java.nio.ByteBuffer should be large enough to accommodate. The FPGA will specify an offset value which will determine where received data will start being written. If the amount of data plus offset exceed the size

```
    of the data array, then the additional data will be discarded.
    The java.nio.ByteBuffer's position and limit are not modified. If
     timeout is non-zero, this call will wait up to timeout ms for
    the FPGA to respond (between packets) before timing out. If
    timeout is zero, this call may block indefinitely. Multiple
    threads receiving on the same channel may result in corrupt data
    or error. This function is thread safe across channels. Returns
    the number of words received to the java.nio.ByteBuffer.

Returns:
The number of words received to the java.nio.ByteBuffer.
```

```
public void reset()

Resets the state of the FPGA and all channels. This is meant to be
    used as an alternative to rebooting if an error occurs while
    sending/receiving. Calling this function while other threads are
    sending or receiving will result in unexpected behavior.

Returns:
Nothing.
```

B.5 Python API

The software interface is provided by bindings for C/C++, Python, and Java. After installation all bindings are available in their respective runtime environments. The API is based on the notion of channels. RIFFA 2.0 can be configured to support between 1 - 12 independent channels. Each channel connects to an IP core and can be addressed by specifying the channel number from the user application. The channels are independent and thread safe. At most one thread should be used to access a single channel.

The Python bindings are used by importing the `riffa` module. You will need Python 2.7 or greater. Below are a couple of complete examples and an API listing.

```
import riffa
import array

fid = 0
channel = 0

data = array.array('I', range(100))
fd = riffa.fpga_open(fid)
riffa.fpga_send(fd, channel, data, 100, 0, True, 0)
riffa.fpga_recv(fd, channel, data, 0)
riffa.fpga_close(fd)
```

This example first opens up FPGA with id 0. It then sends 400 bytes of data (100 4-byte words) to channel 0 with 0 destination offset, 0 timeout, and marks the transfer as last. The IP core on channel 0 is designed to send back some data. So the next call is to receive data from the same channel, up to 400 bytes (that's the size of the data array), with 0 timeout. Note a few things:

- We're using a `array` to hold our data. The `array` type supports the Python buffer protocol interface which means a `memoryview` object can be created from it[1]. A `memoryview` object exposes the underlying memory of the object implementing the buffer protocol interface. This memory is read from and written to directly without copying. Avoiding this copying preserves high performance. Therefore, the `fpga_send` and `fpga_recv` functions only accept data objects that support

[1]The Python buffer protocol interface is a Python 3 design, however it has been back ported to Python 2.7. So using version 2.7 should be fine. However, the Python `array` type was not completely implemented in 2.7. As a result you cannot create a `memoryview` on a Python `array` in 2.7. To support this type, we have exploited another method to support Python `array` types as data arguments in Python 2.7.

a `memoryview`. Python `list` and `tuple` do not support the buffer protocol interface. So you'll want to use `array`, numpy `array` or other types that do.

- The length of words sent is independent of the size of the supplied `data` object or the size of each element in the `data` object. This is meant to force the programmer to think about how many bytes each element uses in the `data` object because the IP core designer will need to know how many and how to interpret that data.
- We're using the same buffer to send data and receive data. This is not required, it just makes for a simpler example.
- The timeout is set to 0. If there's a problem with the IP core logic, a 0 timeout will cause the program to wait forever.
- In practice, you'd want to check the return values to see how much data was sent and received. You'd also probably want some error handling.

Another example is illustrated below that uses the numpy package.

```
import riffa
import numpy

fid = 0
channel = 0

data = numpy.array(range(100))
fd = riffa.fpga_open(fid)
riffa.fpga_send(fd, channel, data, 200, 0, True, 0)
riffa.fpga_recv(fd, channel, data, 0)
riffa.fpga_close(fd)
```

This example is nearly identical to the previous except that it uses a numpy `array`. Since the numpy `array` stores values using 8 bytes per element, we changed the length value to accommodate.

B.5.1 API

The API is provided in the listing below.

```
riffa.fpga_list()

Populates and returns a FpgaInfoList object with information on all
    FPGAs registered in the system or None on error. Print the
    FpgaInfoList object to see the information.

Returns:
A FpgaInfoList object on success, None on error.
```

```
riffa.fpga_open(id)

Initializes the FPGA specified by id. On success, returns an integer
    descriptor for the FPGA. On error, returns None. Each FPGA must
    be opened before any channels can be accessed. Once opened, any
    number of threads can use the descriptor.

id - Identifier for the FPGA (in single FPGA installations, this is
    always 0).

Returns:
An integer descriptor or None.
```

```
riffa.fpga_close(fd)
```

Cleans up memory/resources for the FPGA specified by the fd descriptor.

fd - FPGA descriptor.

Returns:
Nothing.

---

riffa.fpga_send(fd, chnl, data, length, destoff, last, timeout)

fd - FPGA descriptor.
chnl - Channel number over which to communicate (0-11).
data - Object that implements the Python buffer protocol (i.e. can create a memoryview from the object).
length - Length of data to send, in (32 bit) words. Thus a value of 4 means send 16 bytes. Not necessarily the number of elements from data to send.
destoff - Value sent to FPGA core to indicate where to start writing this data. Only the least significant 31 bits are sent (not all 32).
last - If True, this transfer is the last in a sequence of transfers. If False, this transfer is not the last in a sequence of transfers (more transfers to come).
timeout - Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

Sends length words (4 byte words) from data to FPGA channel chnl using the fd descriptor. The data object must implement the Python buffer protocol or an exception will be raised. Note that Python array and numpy array both implement the protocol. The

FPGA channel will be sent length, destoff, and last. You can use Python array slicing (e.g. [m:n] syntax) to create a new array object to send a subset of data. Array slicing will not copy the data on Python buffer protocol capable objects. The value of destoff is used to support sending of data across multiple send transactions. Note that only the low 31 bits of this value are sent. If last is True, the channel should interpret the end of this send as the end of a transaction. If last is False, the channel should wait for additional sends before the end of the transaction. If timeout is non-zero, this call will send data and wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero, this call may block indefinitely. Multiple threads sending on the same channel may result in corrupt data or error. This function is thread safe across channels. Returns the number of words sent.

Returns:
The number of words sent.

---

riffa.fpga_recv(fd, chnl, data, timeout);

fd – FPGA descriptor.
chnl – Channel number over which to communicate (0-11).
data – Object that implements the Python buffer protocol (i.e. can create a memoryview from the object).
timeout – Timeout value in ms. If 0, no timeout is specified. Otherwise, the PC will wait up to timeout ms in between PC/FPGA communications.

Receives data from the FPGA channel chnl to the data object, using the fd descriptor. Just as with the fpga_send function, the data

object must implement the Python buffer protocol or an exception will be raised. The FPGA channel can send any amount of data, so the data array must be large enough to accommodate. The FPGA will specify an offset value which will determine where received data will start being written. If the amount of data plus offset exceed the size of the data array, then the additional data will be discarded. If timeout is non-zero, this call will wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero, this call may block indefinitely. Multiple threads receiving on the same channel may result in corrupt data or error. This function is thread safe across channels. Returns the number of words received to the data array.

Returns:
The number of words received to the data array.

---

riffa.fpga_reset(fd);

Resets the state of the FPGA and all transfers across all channels. This is meant to be used as an alternative to rebooting if an error occurs while sending/receiving. Calling this function while other threads are sending or receiving will result in unexpected behavior.

fd - FPGA descriptor.

Returns:
Nothing.

B.6 Design Tips

When creating a design consider the following tips:

- Start with a simple design. Use the provided example IP core (chnl_tester) on a single channel design to make sure you've setup everything correctly. Then replace the example IP core with your application. It is much easier knowing that RIFFA is setup correctly before attempting to debug your entire design.

- Reboot after changing the FPGA design (i.e. after flashing via JTag). The operating system needs to probe the PCIe device at boot time and reserve PCIe device space. In theory, most operating systems support hot plugging PCIe devices. But we have not be able to get this to work reliably ourselves.

- Check that your operating system recognizes the FPGA as a PCIe device after it boots. This is the first thing to check. In Linux you can run `lspci` and look for the FPGA in the list. You should see the FPGA listed as a PCI Memory device. You can also check the kernel log, type `dmesg` to see if the RIFFA driver loaded correctly. There should be entries in the log with the prefix "riffa". In Windows you can check the Device Manager. You should see the FPGA listed as a RIFFA Device. If you've installed DgbView and configured it to capture boot information, you should find entries in the log with the prefix "riffa".

- Routing a design that makes timing constraints can be difficult when using multiple RIFFA channels (4+). We have successfully implemented designs with all 12 channels on all our boards at all the data bit widths (32, 64, 128). We found that using ISE's Smart Explorer was necessary to find the correct set of options for some designs.

- To achieve the highest bandwidth from the PC to your IP core (and vice versa), you'll want to drive the RIFFA channel with the PCIe interface clock. This is the clock RIFFA uses to read/write data from the PCIe Endpoint. For high numbers of lane links or 5.0 GT/s link speeds, this will often be 250 MHz. You don't need to run the rest of your design off this clock, but consider driving the RX and TX channel ports using this clock so you can keep up with RIFFA.

- Follow the design suggestions and examples on the software bindings pages. The examples shown avoid unnecessary copying of data (when possible).

- Avoid using Intel Z77 Ivy Bridge motherboards with RIFFA. This motherboard architecture causes numerous problems with the Xilinx PCIe Endpoints.

- Monitor your kernel system log (or DbgView on Windows). This is where RIFFA outputs debug information.

- Remember that VHDL and Verilog are compatible with little endian number ordering. The C/C++ and Python bindings will use the system processor endianness (which is also little endian for Intel/AMD). Java however encodes numbers in big endian ordering. If using the Java bindings, data sent and received may need byte swapping. The ByteBuffer classes will encode/decode numeric values in big endian format. Byte swapping is most efficiently done on the FPGA as data is being transferred.

B.7 Design Guide - Avnet Xilinx S6LX150t - ISE

This is a step by step guide to building a RIFFA 2.0 reference design for an Avnet Xilinx Spartan-6 LX150t development board using ISE. Though it is likely that this guide will work for other Spartan-6 based FPGA development boards.

RIFFA 2.0 provides a simple to use interface for communicating between a workstation and FPGA cores. It uses a Xilinx PCIe Endpoint IP core to drive the transceivers. The PCIe Endpoint core for Spartan 6 FPGAs is the Spartan 6 Integrated Block for PCI Express. This core is licensed by the Xilinx End User License Agreement and is provided with the Xilinx ISE Design suite with no additional charge. A prebuilt design is provided and ready for download to your Avnet Xilinx Spartan-6 LX150t board. Building your own RIFFA 2.0 design requires generating the PCIe Endpoint core and then merging it with the RIFFA 2.0 source HDL.

To create a RIFFA 2.0 design with ISE:

1. Use Xilinx Coregen to generate the PCIe Endpoint core.
2. Combine the PCIe Endpoint core's source HDL with the RIFFA 2.0 HDL.
3. Create a project in Xilinx ISE with the combined source HDL and.ucf.
4. Synthesize and implement.

Detailed instructions on how to do each step follow.

1. Use Xilinx Coregen to generate the PCIe Endpoint core. Use Coregen to generate Verilog source for the Spartan 6 Integrated Block for PCI Express ver. 2.4. This is the latest production version of the core at the time of this writing.

Start Coregen and make sure to set the project settings to generate Verilog code for the XC6SLX150t-3FGG676. Use the Coregen wizard to generate the core. Unless otherwise described, the default values on each wizard screen should be left as they are presented.

On the first screen, pictured in Figure B.3, you will see that you have no selections to make for lane width or link speed as this core only supports one lane at 2.5 GT/s. This results in a 32 bit interface and a 62.5 MHz interface frequency clock. RIFFA 2.0 supports 32, 64, and 128 bit interfaces. You can use whatever interface frequency the

Table B.2. Maximum theoretical bandwidths for PCIe 1.0 and PCIe 2.0.

Gen1 (2.5 GT/s):	Gen1 (2.5 GT/s):
x1 = 250 MB/s	x1 = 500 MB/s
x2 = 500 MB/s	x2 = 100 MB/s
x4 = 1000 MB/s	x4 = 2000 MB/s
x8 = 2000 MB/s	x8 = 4000 MB/s

options allow. We keep the default component name in our reference design. Table B.2 lists maximum theoretical bandwidths for PCIe 1.0 and PCIe 2.0.

On the next screen, pictured in Figure B.4, make sure only Bar0 is selected and is set to a size of 1 KB. You will need to deselect Bar2.

On the next screen, pictured in Figure B.5, select Performance Level High. Additionally, you will want to set the Max Payload Size to the maximum value offered. These changes are not necessary for RIFFA 2.0 to function. They are required to achieve maximum performance.

On this last screen, pictured in Figure B.6, select the development board that you are using. If your development board is not in the list, you will need to know the PCIe Block location for your part-package combination. Additional modifications to the generated .ucf may also be necessary if your board is not in the list. As the Avnet Xilinx Spartan-6 LX150t development board is not in the list, the RIFFA 2.0 reference design comes with a configured .ucf file. Set the Reference Clock Frequency to 125 MHz. Then complete the wizard and generate the core.

2. Combine the PCIe Endpoint core's source HDL with the RIFFA 2.0 HDL. Core-gen will produce a directory structure similar to what is pictured in Figure B.7. Once completed, combine all the source HDL files from the source directory with the RIFFA 2.0 HDL files from the distribution into a new directory of your choosing. Also, into this new directory, copy the top level and adapter module HDL files for this board from the

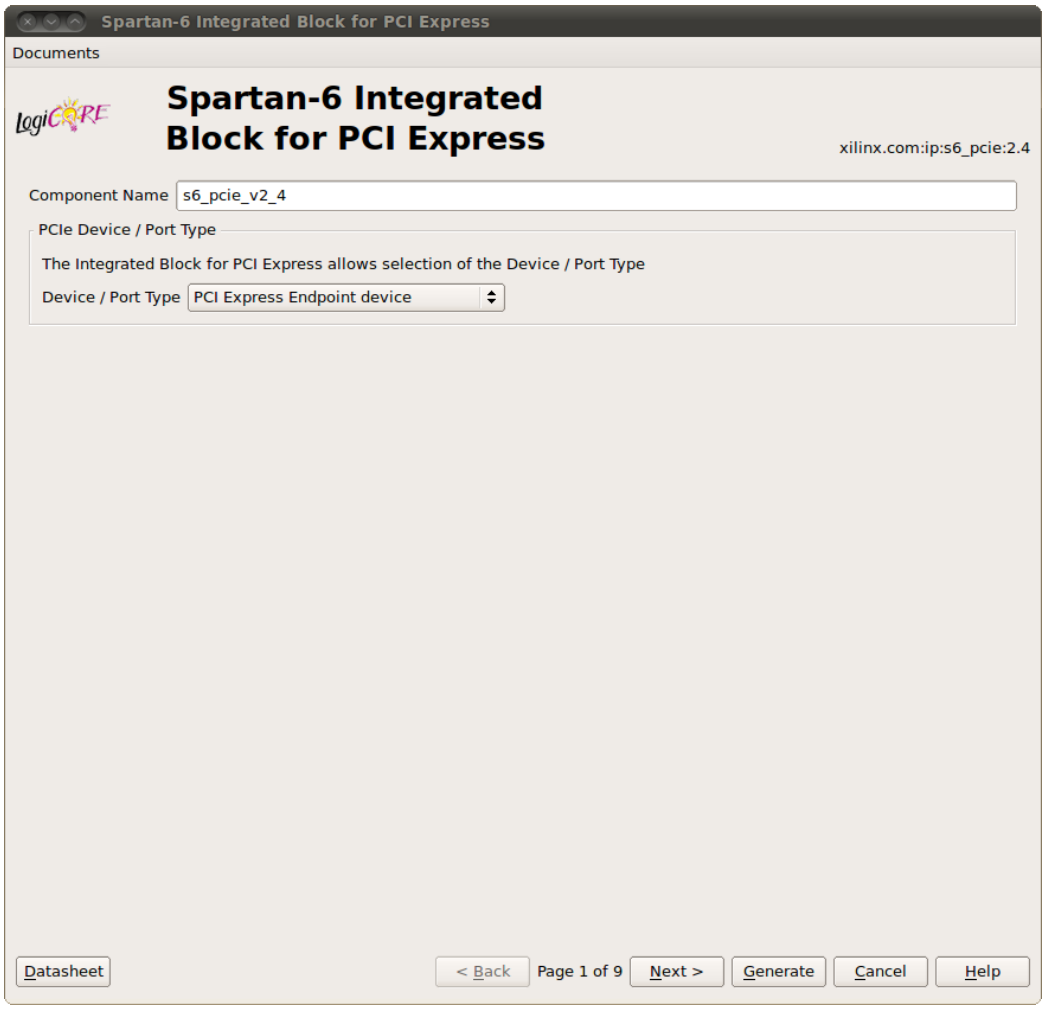


Figure B.3. Xilinx Coregen wizard screen.

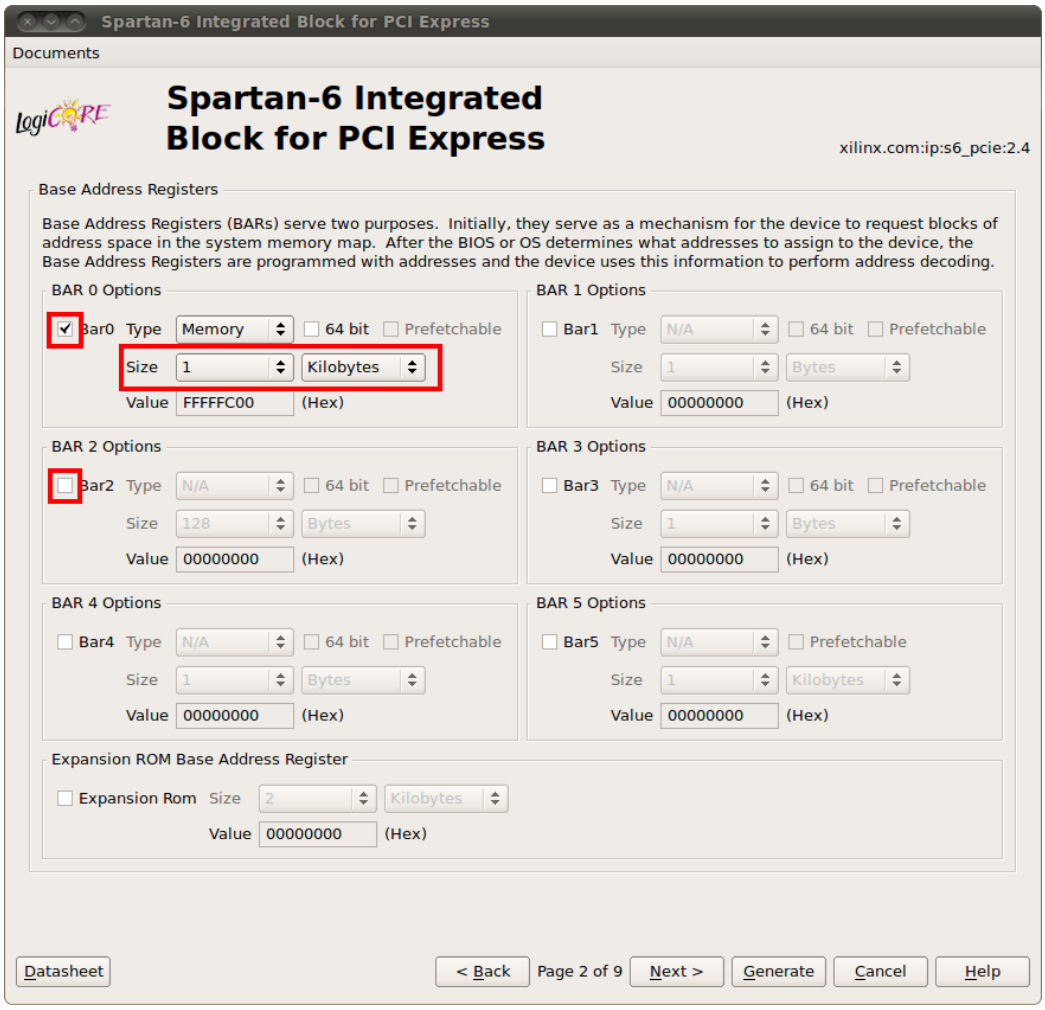


Figure B.4. Xilinx Coregen wizard screen.

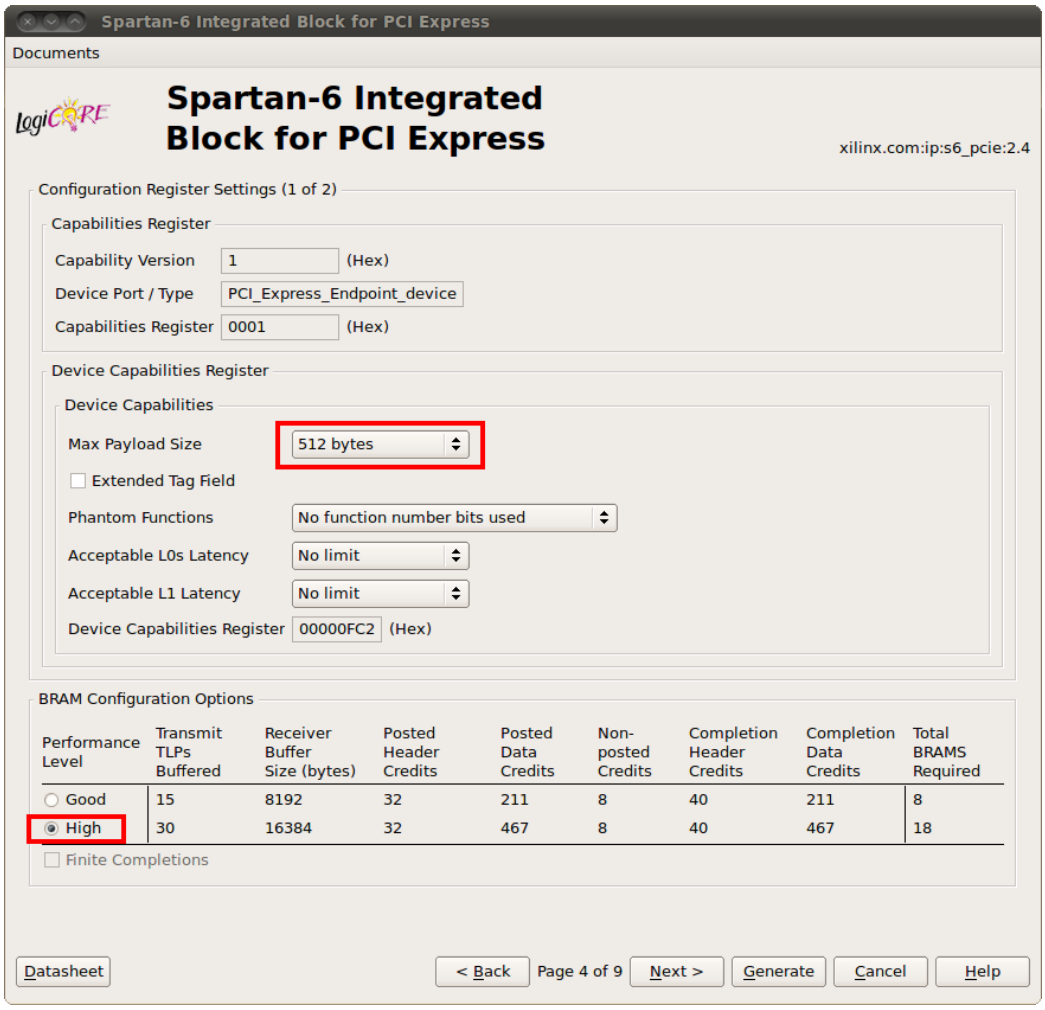


Figure B.5. Xilinx Coregen wizard screen.

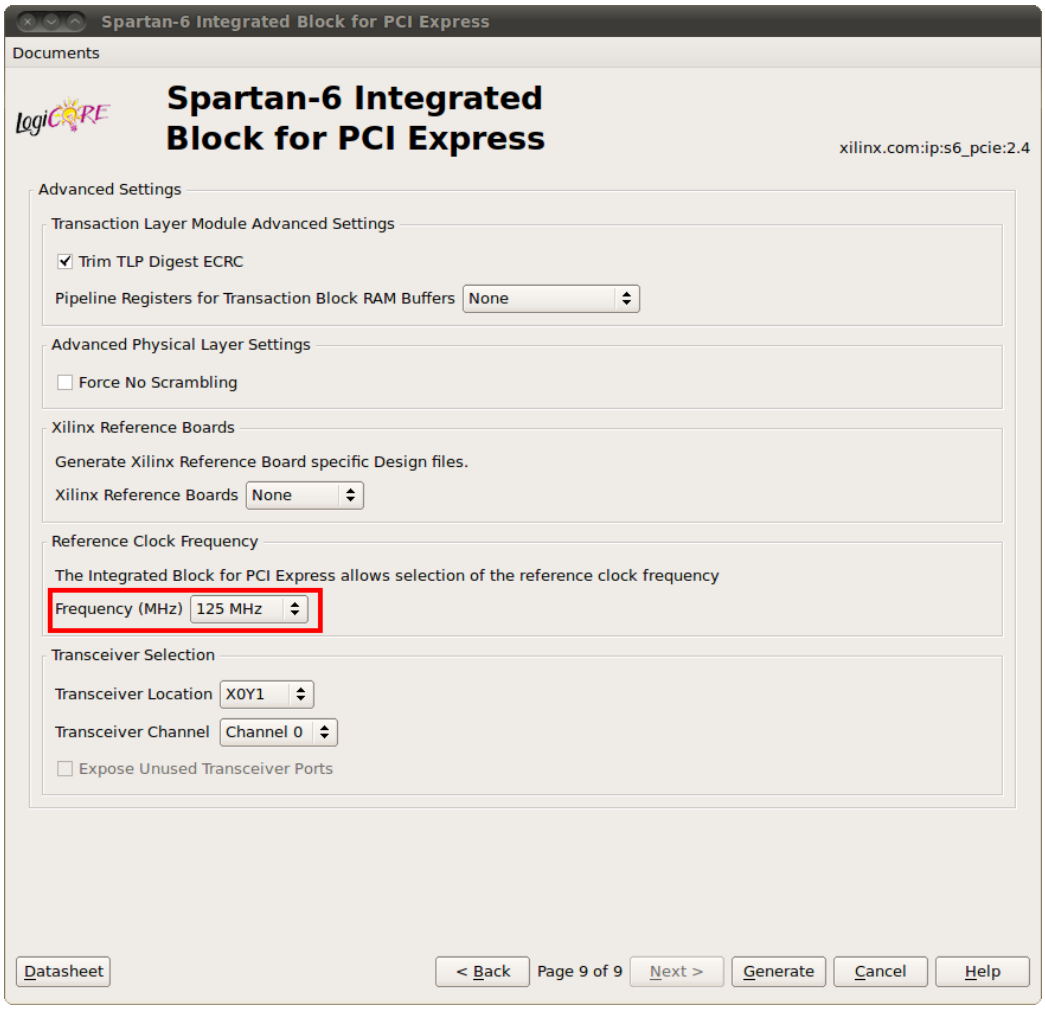


Figure B.6. Xilinx Coregen wizard screen.

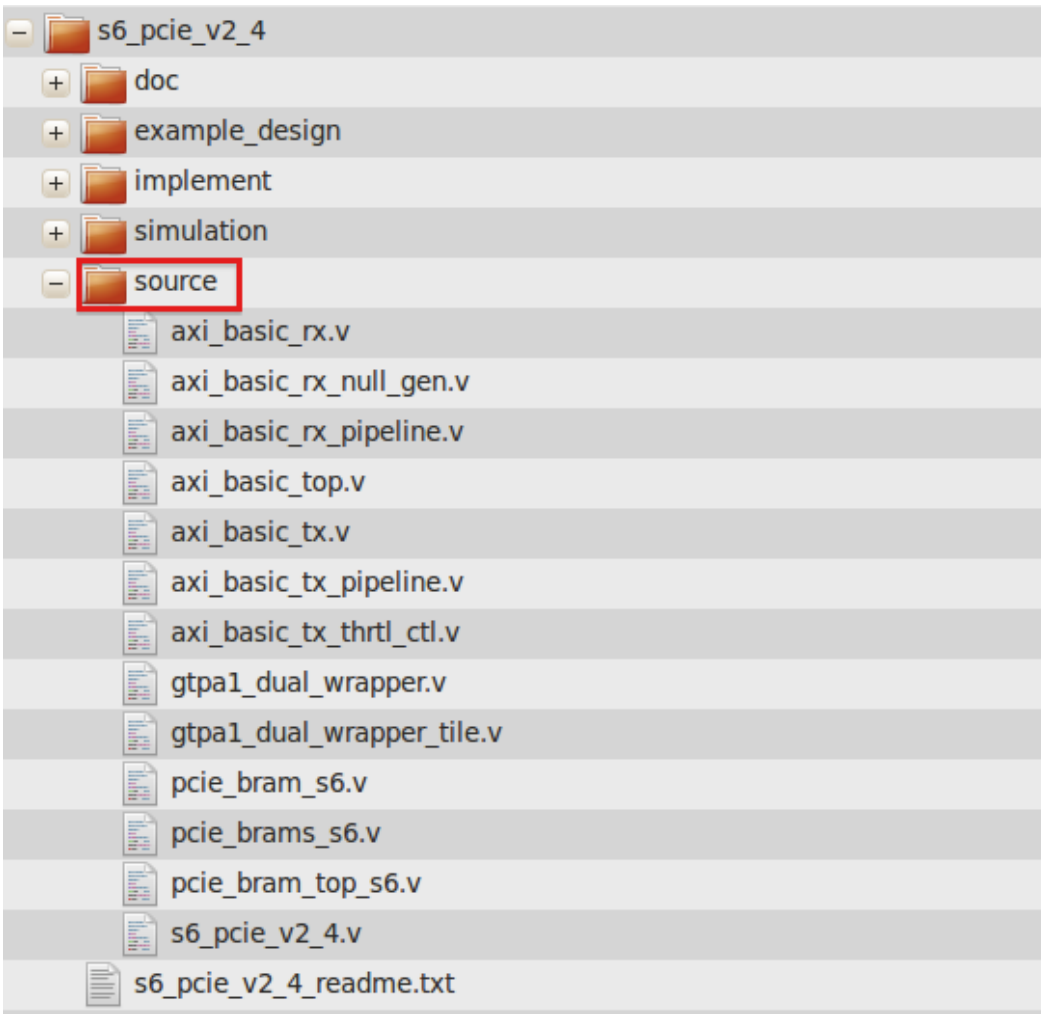


Figure B.7. File tree listing.

RIFFA 2.0 distribution. Lastly use the .ucf file from the RIFFA 2.0 distribution. Do not use the top level module or .ucf from the example_design directory as they do not have the necessary modifications to work correctly.

3. Create a project in ISE with the combined HDL and.ucf. I expect you know how to create a new project in ISE. So I won't provide step by step instructions.

4. Synthesize and implement. At this point, if you attempt to synthesize you will encounter an error: C_NUM_CHNL is not defined. This is intentional. It is done to

get you to open the RIFFA 2.0 adapter module file and edit it as needed. There are instructions in that file. However, all you need to do is uncomment the line that defines the C_NUM_CHNL parameter, set it to the number of channels you need (1-12), and you should be able to implement the design completely.

The adapter module instantiates a chnl_tester module for each channel. Sample user application software in the RIFFA 2.0 distribution can be used to send and receive data to/from the chnl_tester modules. The chnl_tester is meant to be an example. Your design will need to replace the chnl_tester modules with your own modules. You may also need to modify the .ucf, top level, and RIFFA adapter modules to bring in additional signals as dictated by your design.

B.8 Design Guide - Xilinx ML605 - ISE

This is a step by step guide to building a RIFFA 2.0 reference design for a Xilinx Virtex-6 ML605 development board using ISE. Though it is likely that this guide will work for other Virtex-6 based FPGA development boards.

RIFFA 2.0 provides a simple to use interface for communicating between a workstation and FPGA cores. It uses a Xilinx PCIe Endpoint IP core to drive the transceivers. The PCIe Endpoint core for Spartan 6 FPGAs is the Virtex 6 Integrated Block for PCI Express. This core is licensed by the Xilinx End User License Agreement and is provided with the Xilinx ISE Design suite with no additional charge. A prebuilt design is provided and ready for download to your Xilinx ML605 board. Building your own RIFFA 2.0 design requires generating the PCIe Endpoint core and then merging it with the RIFFA 2.0 source HDL.

To create a RIFFA 2.0 design with ISE:

1. Use Xilinx Coregen to generate the PCIe Endpoint core.
2. Combine the PCIe Endpoint core's source HDL with the RIFFA 2.0 HDL.

3. Create a project in Xilinx ISE with the combined source HDL and.ucf.

4. Synthesize and implement.

Detailed instructions on how to do each step follow.

1. Use Xilinx Coregen to generate the PCIe Endpoint core. Use Coregen to generate Verilog source for the Virtex 6 Integrated Block for PCI Express ver. 2.5. This is the latest production version of the core at the time of this writing.

Start Coregen and make sure to set the project settings to generate Verilog code for the XC6VLX240t-1FFG1156. Use the Coregen wizard to generate the core. Unless otherwise described, the default values on each wizard screen should be left as they are presented.

On the first screen, pictured in Figure B.8, select the desired lane width and link speed. RIFFA 2.0 supports a 32, 64, and 128 bit interface. So any lane width and link speed selection you make will be supported. You can use any interface frequency the options allow. We keep the default component name. Table B.2 lists maximum theoretical bandwidths for PCIe 1.0 and PCIe 2.0.

On the next screen, pictured in Figure B.9, make sure only Bar0 is selected and is set to a size of 1 KB. You will need to deselect Bar2.

On the next screen, pictured in Figure B.10, select Buffering Optimized for Bus Mastering Applications and Performance Level High. Additionally, set the Max Payload Size to the maximum value offered. These changes are not necessary for RIFFA 2.0 to function. They are required to achieve maximum performance.

On the next screen, pictured in Figure B.11, select the development board that you are using. If your board is not in the list, you will need to know the PCIe Block location for your part-package combination. Additional modifications to the generated .ucf may also be necessary if your board is not in the list.

Figure B.8. Xilinx Coregen wizard screen.

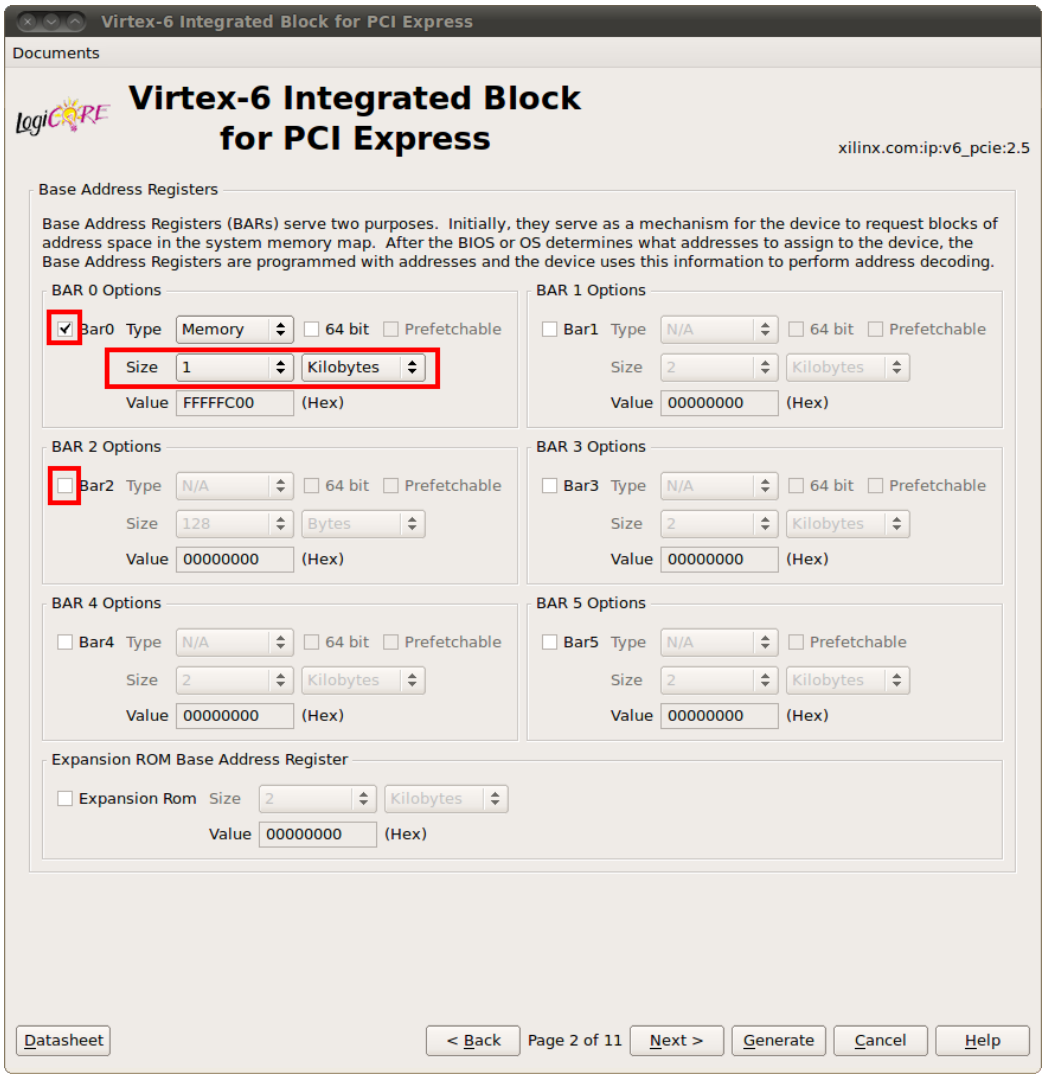


Figure B.9. Xilinx Coregen wizard screen.

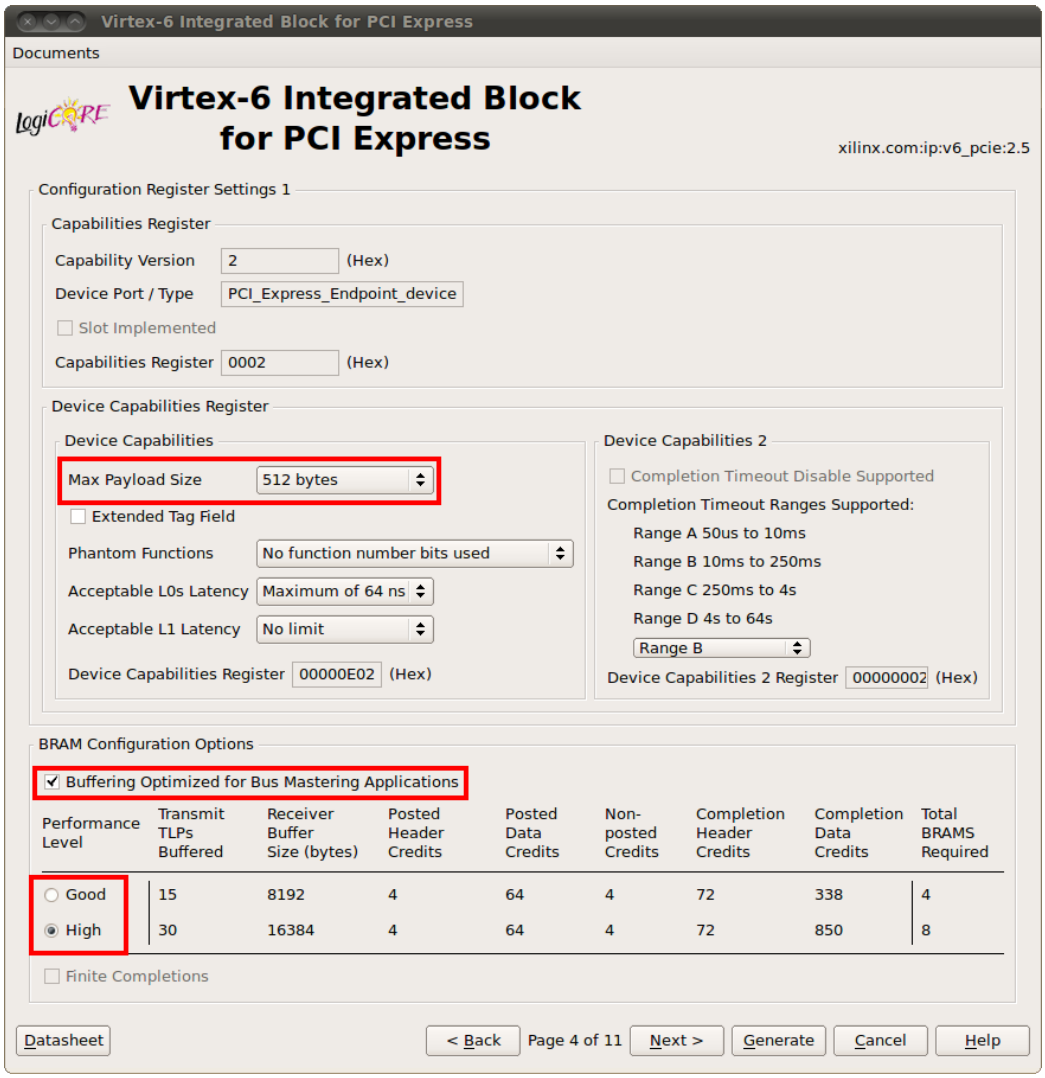


Figure B.10. Xilinx Coregen wizard screen.

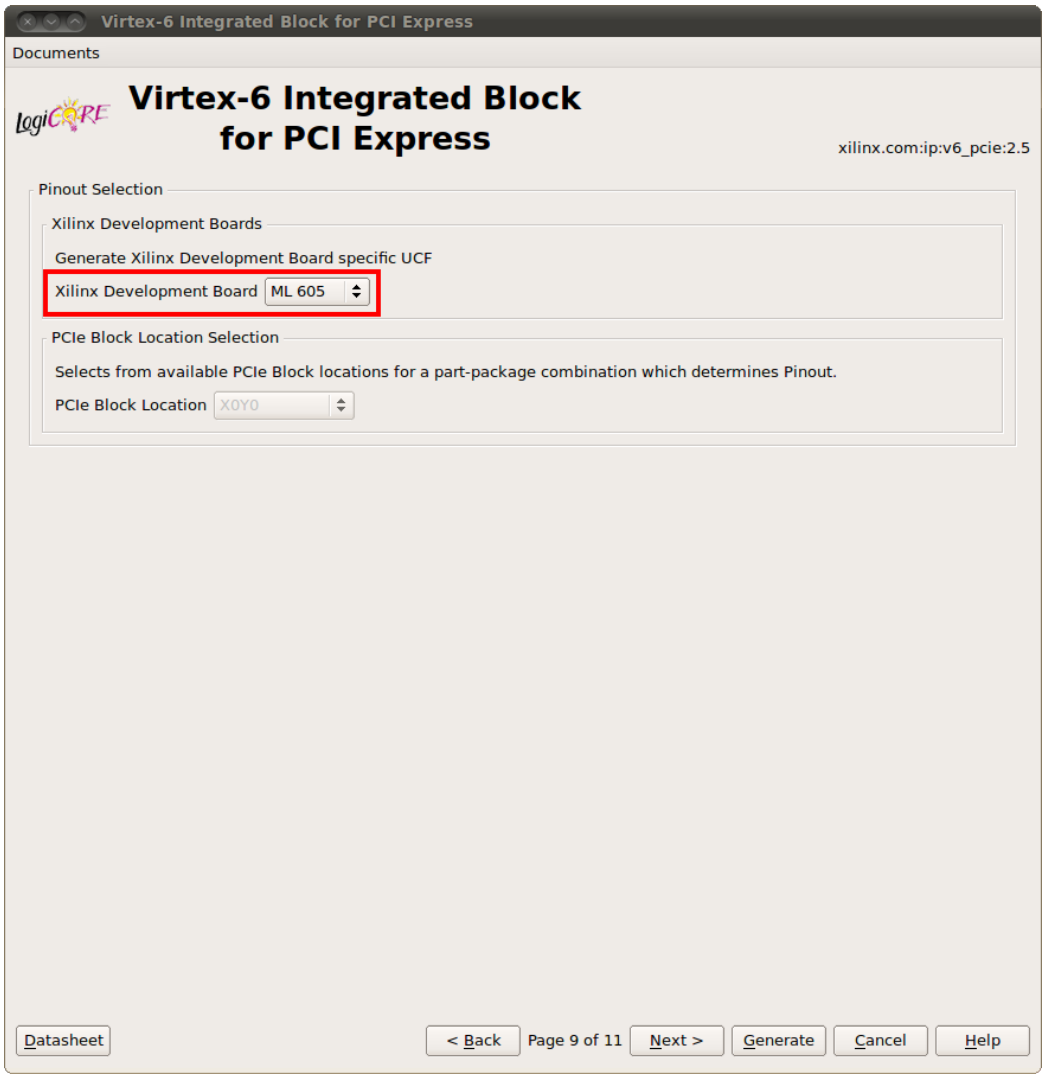


Figure B.11. Xilinx Coregen wizard screen.

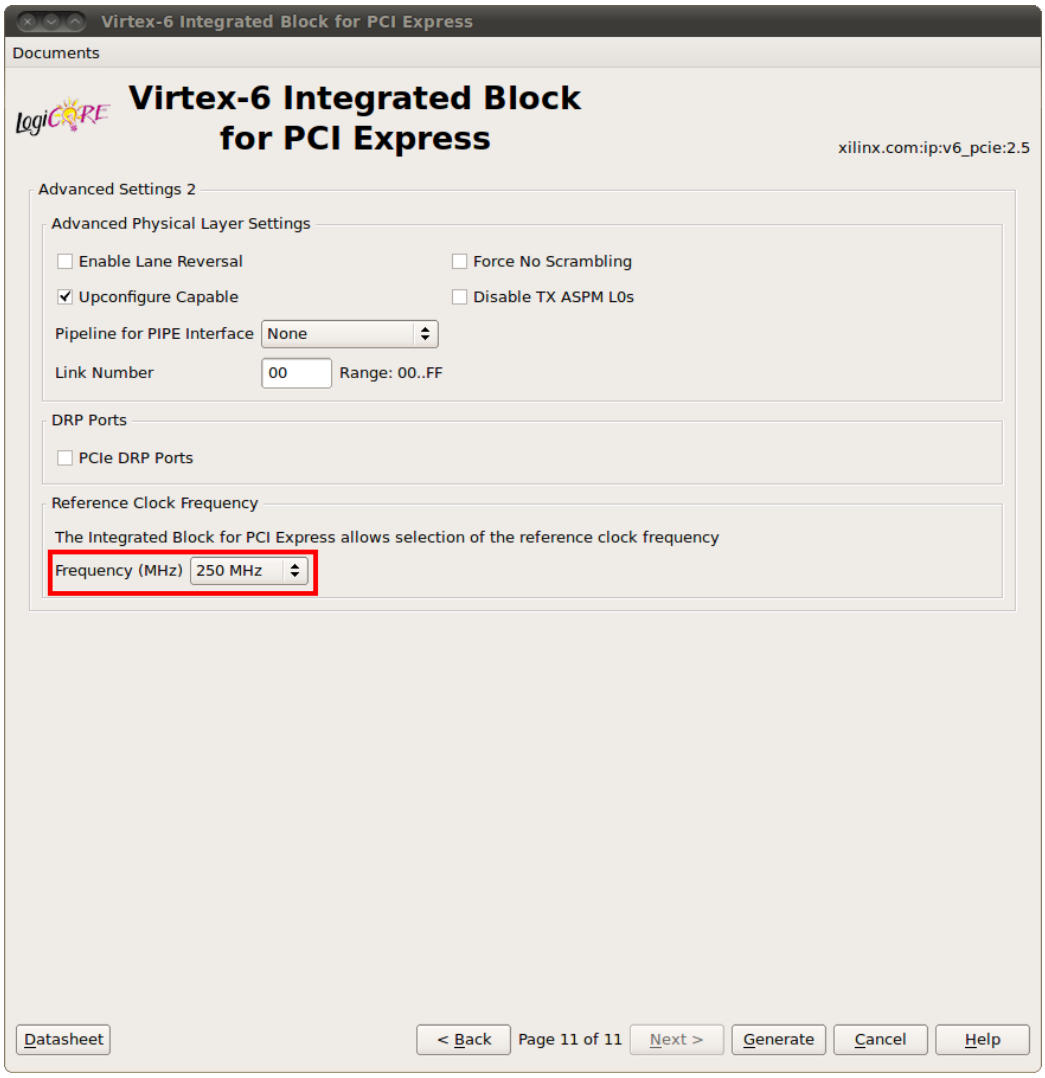


Figure B.12. Xilinx Coregen wizard screen.

On this last screen, pictured in Figure B.12, set the Reference Clock Frequency to 250 MHz. This is the recommended setting from Xilinx document XTP044. Then complete the wizard and generate the core.

2. Combine the PCIe Endpoint core's source HDL with the RIFFA 2.0 HDL. Coregen will produce a directory structure similar to what is pictured in Figure B.13. Once completed, combine all the source HDL files from the source directory with the RIFFA

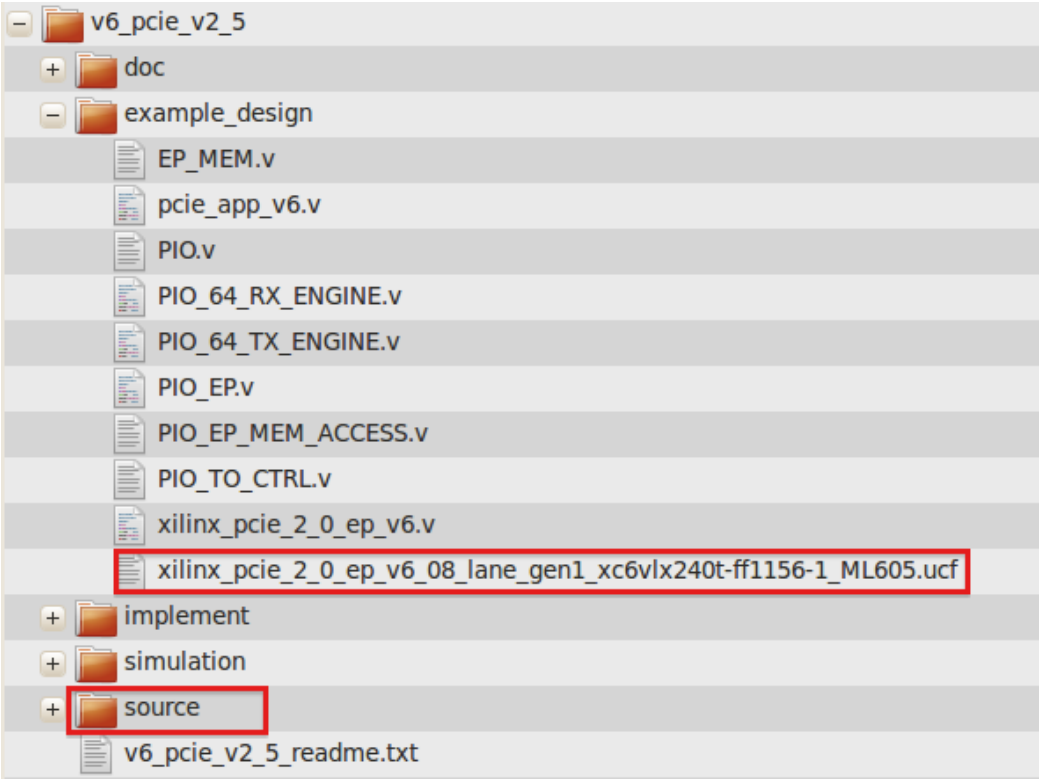


Figure B.13. File tree listing.

2.0 HDL files from the distribution into a new directory of your choosing. Also, into this new directory, copy the top level and adapter module HDL files for this board from the RIFFA 2.0 distribution. Lastly use the .ucf file from the RIFFA 2.0 distribution. Do not use the top level module or .ucf from the example_design directory as they do not have the necessary modifications to work correctly.

3. Create a project in ISE with the combined HDL and.ucf. I expect you know how to create a new project in ISE. So I won't provide step by step instructions.

4. Synthesize and implement. At this point, if you attempt to synthesize you will encounter an error: C_NUM_CHNL is not defined. This is intentional. It is done to get you to open the RIFFA 2.0 adapter module file and edit it as needed. There are

instructions in that file. However, all you need to do is uncomment the line that defines the C_NUM_CHNL parameter, set it to the number of channels you need (1-12), and you should be able to implement the design completely.

The adapter module instantiates a chnl_tester module for each channel. Sample user application software in the RIFFA 2.0 distribution can be used to send and receive data to/from the chnl_tester modules. The chnl_tester is meant to be an example. Your design will need to replace the chnl_tester modules with your own modules. You may also need to modify the .ucf, top level, and RIFFA adapter modules to bring in additional signals as dictated by your design.

B.9 Design Guide - Xilinx VC707 - ISE

This is a step by step guide to building a RIFFA 2.0 reference design for a Xilinx Virtex-7 VC707 development board using ISE. Though it is likely that this guide will work for other 7 Series based FPGA development boards.

RIFFA 2.0 provides a simple to use interface for communicating between a workstation and FPGA cores. It uses a Xilinx PCIe Endpoint IP core to drive the transceivers. The PCIe Endpoint core for 7 Series FPGAs is the 7 Series Integrated Block for PCI Express. This core is licensed by the Xilinx End User License Agreement and is provided with the Xilinx ISE Design suite with no additional charge. A prebuilt design is provided and ready for download to your Xilinx VC707 board. Building your own RIFFA 2.0 design requires generating the PCIe Endpoint core and then merging it with the RIFFA 2.0 source HDL.

To create a RIFFA 2.0 design with ISE:

1. Use Xilinx Coregen to generate the PCIe Endpoint core.
2. Combine the PCIe Endpoint core's source HDL with the RIFFA 2.0 HDL.
3. Create a project in Xilinx ISE with the combined source HDL and.ucf.

4. Synthesize and implement.

Detailed instructions on how to do each step follow.

1. Use Xilinx Coregen to generate the PCIe Endpoint core. Use Coregen to generate Verilog source for the 7 Series Integrated Block for PCI Express ver. 1.8. This is the latest production version of the core at the time of this writing.

Start Coregen and make sure to set the project settings to generate Verilog code for the XC7VX485t-2FFG1761C. Use the Coregen wizard to generate the core. Unless otherwise described, the default values on each wizard screen should be left as they are presented.

On the first screen, pictured in Figure B.14, select the desired lane width and link speed. RIFFA 2.0 supports a 32, 64, and 128 bit interface. So any lane width and link speed selection you make will be supported. You can use any interface frequency the options allow. We keep the default component name. Table B.2 lists maximum theoretical bandwidths for PCIe 1.0 and PCIe 2.0.

On the next screen, pictured in Figure B.15, make sure only Bar0 is selected and is set to a size of 1 KB.

On the next screen, pictured in Figure B.16, select Buffering Optimized for Bus Mastering Applications and Performance Level High. Additionally, set the Max Payload Size to the maximum value offered. These changes are not necessary for RIFFA 2.0 to function. They are required to achieve maximum performance.

On this last screen, pictured in Figure B.17, select the development board that you are using. If your board is not in the list, you will need to know the PCIe Block location for your part-package combination. Additional modifications to the generated .ucf may also be necessary if your board is not in the list. Then complete the wizard and generate the core.

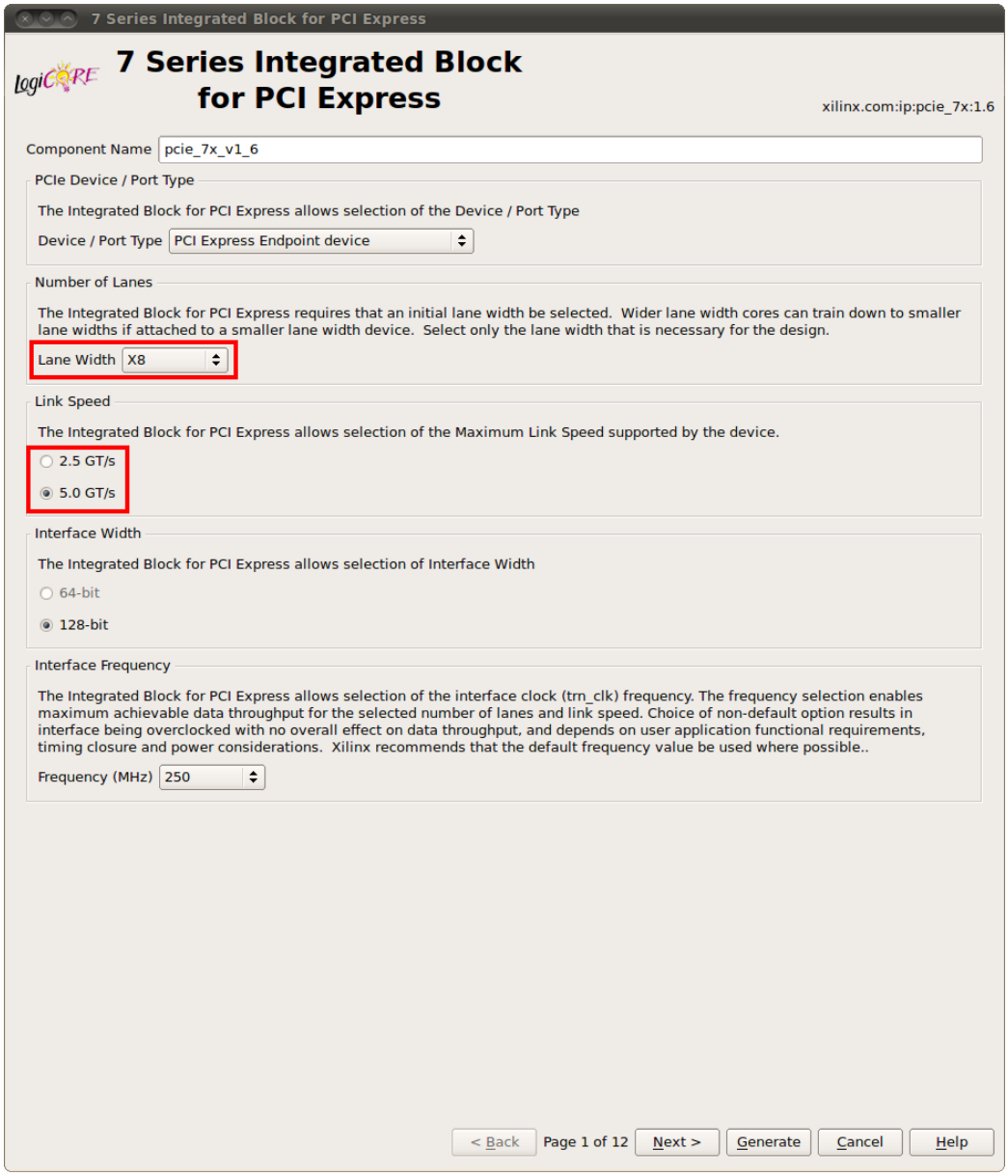


Figure B.14. Xilinx Coregen wizard screen.

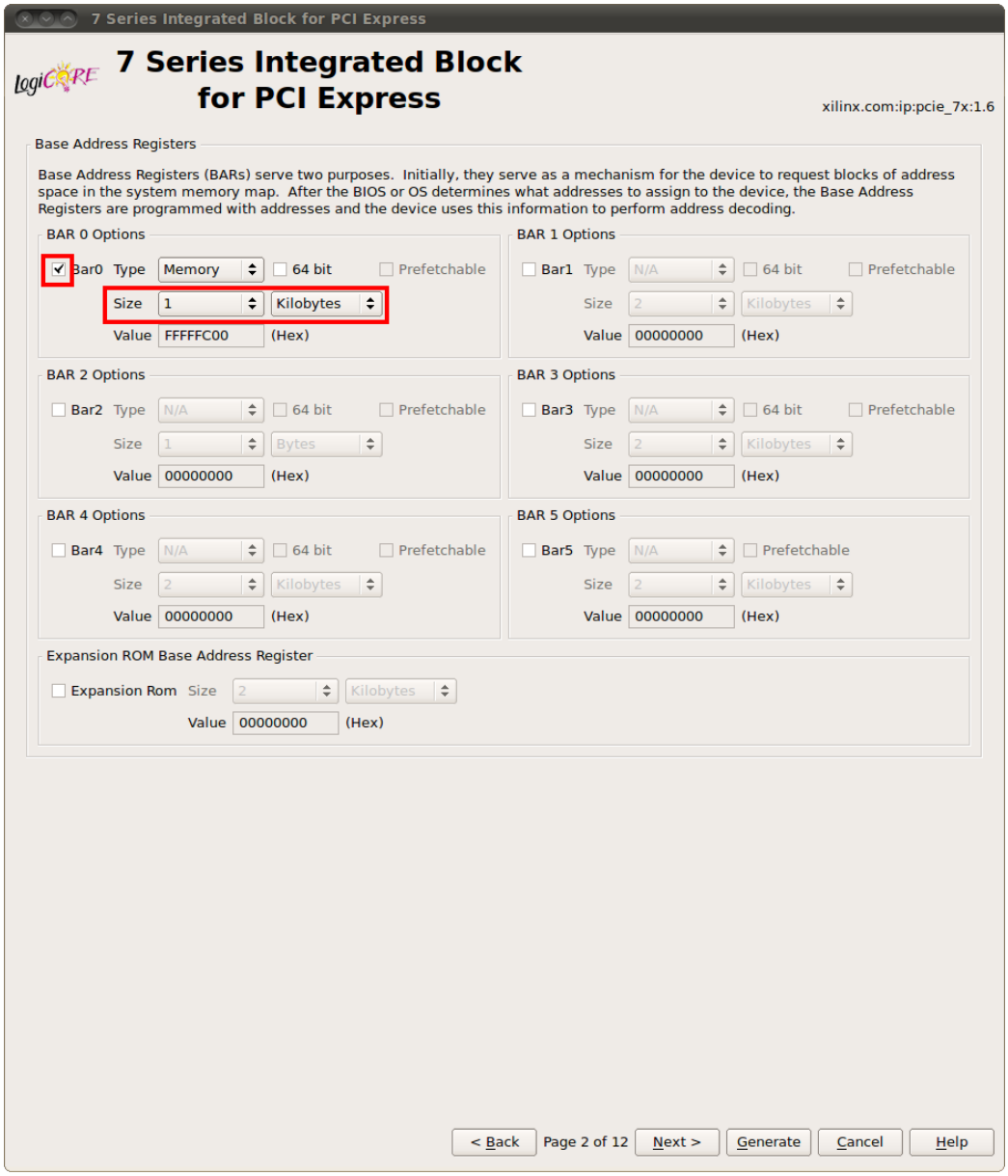


Figure B.15. Xilinx Coregen wizard screen.

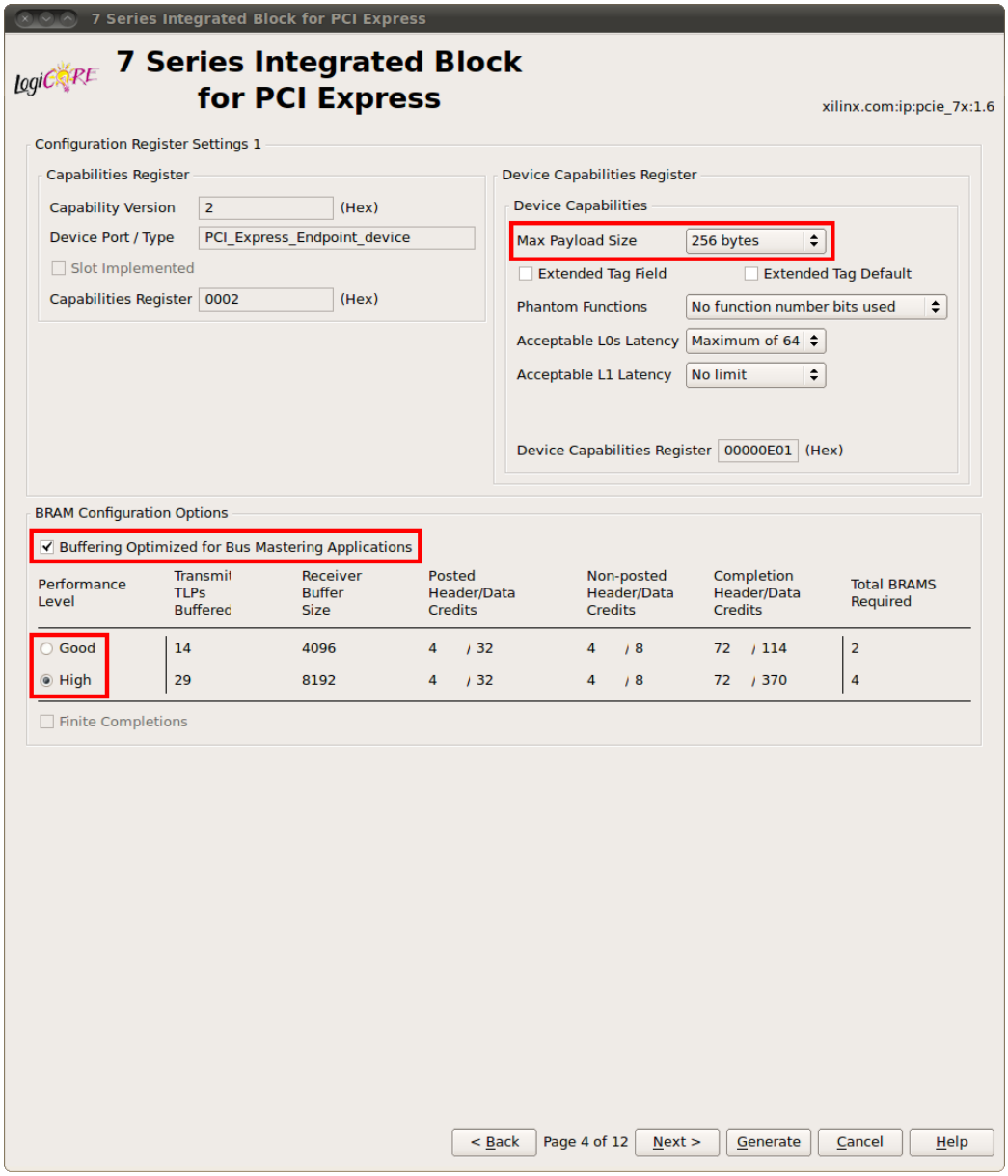


Figure B.16. Xilinx Coregen wizard screen.

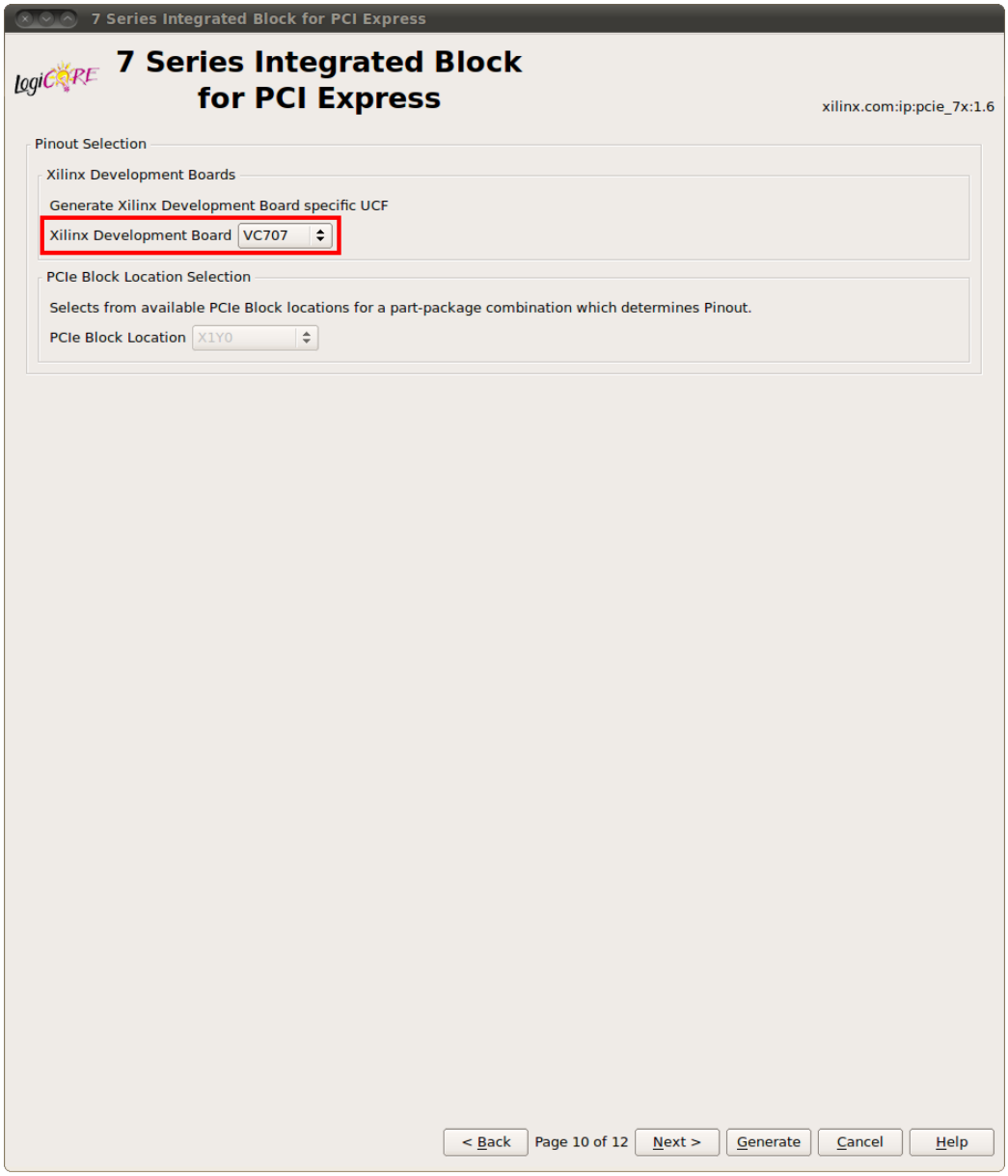


Figure B.17. Xilinx Coregen wizard screen.

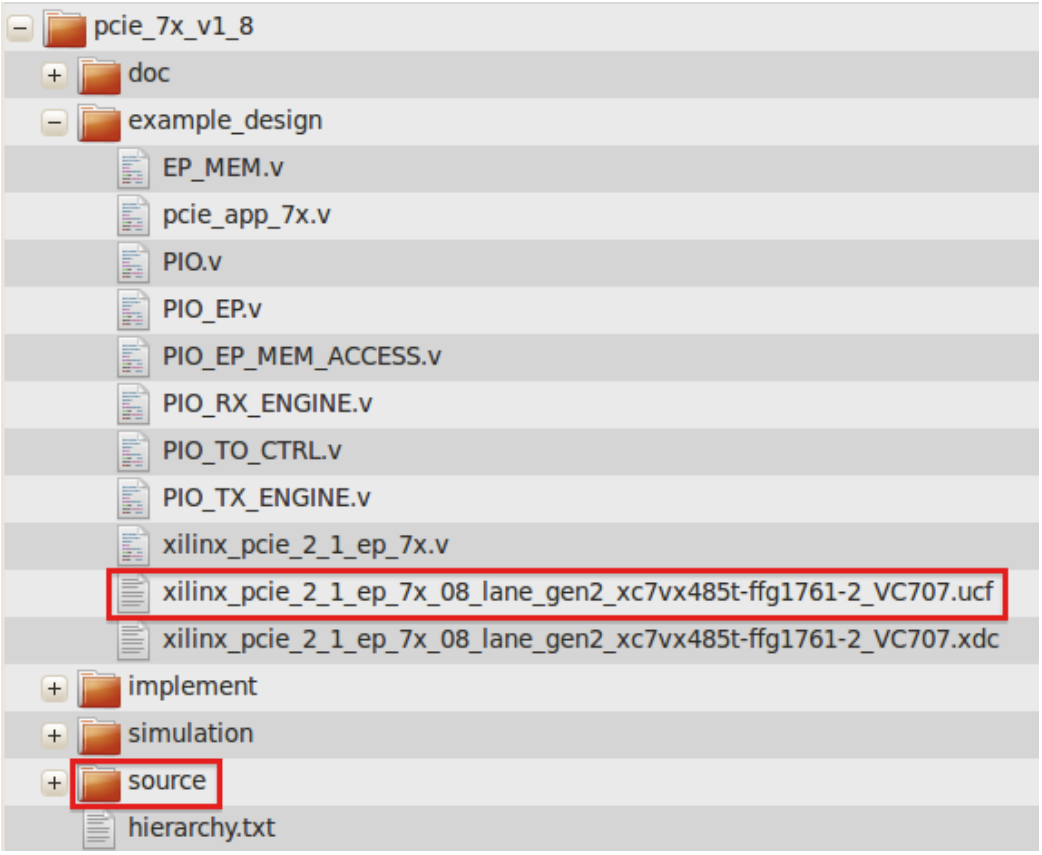


Figure B.18. File tree listing.

2. Combine the PCIe Endpoint core's source HDL with the RIFFA 2.0 HDL. Core-gen will produce a directory structure similar to what is pictured in Figure B.18. Once completed, combine all the source HDL files from the source directory with the RIFFA 2.0 HDL files from the distribution into a new directory of your choosing. Also, into this new directory, copy the top level and adapter module HDL files for this board from the RIFFA 2.0 distribution. Lastly use the .ucf file from the RIFFA 2.0 distribution. Do not use the top level module or .ucf from the example_design directory as they do not have the necessary modifications to work correctly.

3. Create a project in ISE with the combined HDL and.ucf. I expect you know how to create a new project in ISE. So I won't provide step by step instructions.

4. Synthesize and implement. At this point, if you attempt to synthesize you will encounter an error: C_NUM_CHNL is not defined. This is intentional. It is done to get you to open the RIFFA 2.0 adapter module file and edit it as needed. There are instructions in that file. However, all you need to do is uncomment the line that defines the C_NUM_CHNL parameter, set it to the number of channels you need (1-12), and you should be able to implement the design completely.

The adapter module instantiates a chnl_tester module for each channel. Sample user application software in the RIFFA 2.0 distribution can be used to send and receive data to/from the chnl_tester modules. The chnl_tester is meant to be an example. Your design will need to replace the chnl_tester modules with your own modules. You may also need to modify the .ucf, top level, and RIFFA adapter modules to bring in additional signals as dictated by your design.

B.10 Design Guide - Xilinx VC707 - Vivado

This is a step by step guide to building a RIFFA 2.0 reference design for a Xilinx Virtex-7 VC707 development board using Vivado. Though it is likely that this guide will work for other 7 Series based FPGA development boards.

RIFFA 2.0 provides a simple to use interface for communicating between a workstation and FPGA cores. It uses a Xilinx PCIe Endpoint IP core to drive the transceivers. The PCIe Endpoint core for 7 Series FPGAs is the 7 Series Integrated Block for PCI Express. This core is licensed by the Xilinx End User License Agreement and is provided with the Xilinx Vivado Design suite with no additional charge. A prebuilt design is provided and ready for download to your Xilinx VC707 board. Building your own RIFFA 2.0 design requires generating the PCIe Endpoint core and then merging it with the RIFFA 2.0 source HDL.

To create a RIFFA 2.0 design with Vivado:

1. Create a project in Xilinx Vivado.
2. Use Vivado to generate the PCIe Endpoint core.
3. Add the RIFFA 2.0 HDL as design sources.
4. Synthesize and implement.

Detailed instructions on how to do each step follow.

1. Create a project in Xilinx Vivado. I expect you know how to create a new RTL based project in Vivado for your development board. So I won't provide step by step instructions.

2. Use Vivado to generate the PCIe Endpoint core. Select IP Catalog and double click on the 7 Series Integrated Block for PCI Express core. Version 2.1 is the latest production version of the core at the time of this writing. Figure B.19 shows the Vivado IDE. This will begin customization of the IP. Unless otherwise described, the default values on each wizard screen should be left as they are presented.

On the first screen, pictured in Figure B.20, set the desired lane width and link speed. RIFFA 2.0 supports a 32, 64, and 128 bit interface. So any lane width and link speed selection you make will be supported. You can use any interface frequency the options allow. The reference design expects the component name specified below. Leave the reference clock frequency set to 100 MHz. Set the Xilinx Development Board to VC707 and set Silicon Revision to GES and Production. Table B.2 lists maximum theoretical bandwidths for PCIe 1.0 and PCIe 2.0.

On the next screen, pictured in Figure B.21, make sure only Bar0 is selected and is set to a size of 1 KB.

On the next screen, pictured in Figure B.22, set Performance Level to High. Additionally, set the Max Payload Size to the maximum value offered. These changes

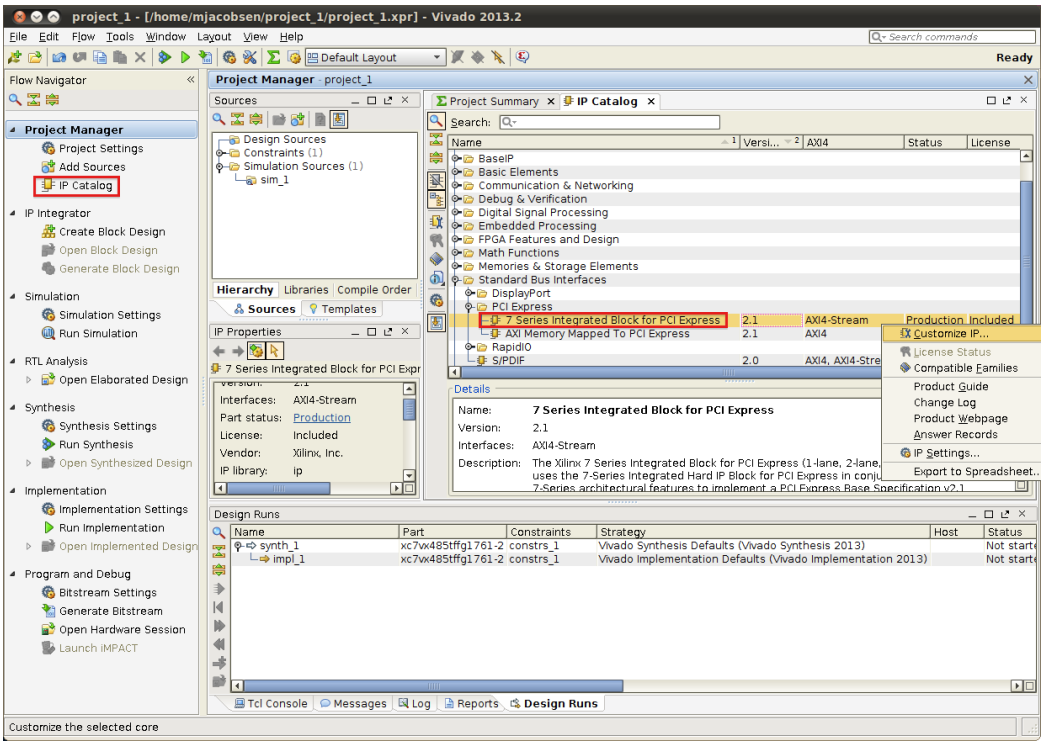


Figure B.19. Xilinx Vivado screen.

are not necessary for RIFFA 2.0 to function. They are required to achieve maximum performance.

Then complete the wizard and generate the core. When prompted with the dialog pictured in Figure B.23, click Generate.

In Vivado you must open the IP Example Design before you can use the PCIe Endpoint. This will create a new Vivado project with the PCIe Endpoint Example Design and open another Vivado instance with this new project. See Figure B.24.

When prompted, specify a location of your choosing for the new Vivado project. Figure B.25 illustrates this dialog.

3. Add the RIFFA 2.0 HDL as design sources. The Example Application Vivado project will be your design project. Before adding the RIFFA 2.0 HDL files, you need to remove the existing example application HDL files. Select the following files and remove

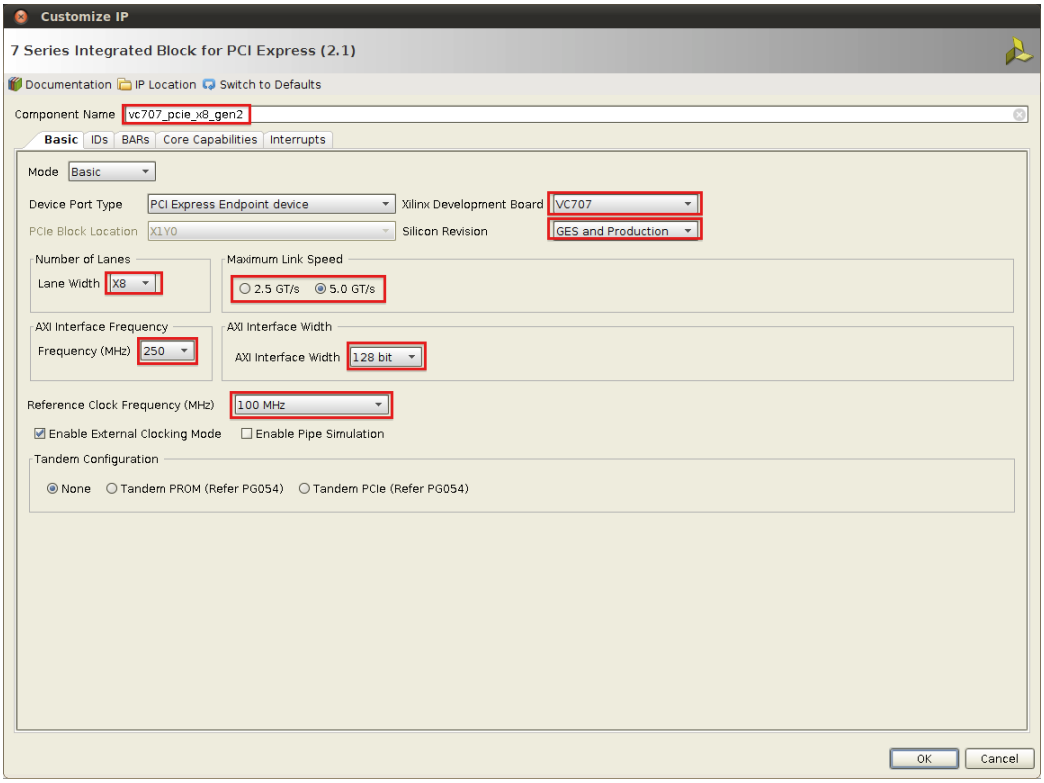


Figure B.20. Xilinx Vivado IP Catalog wizard screen.

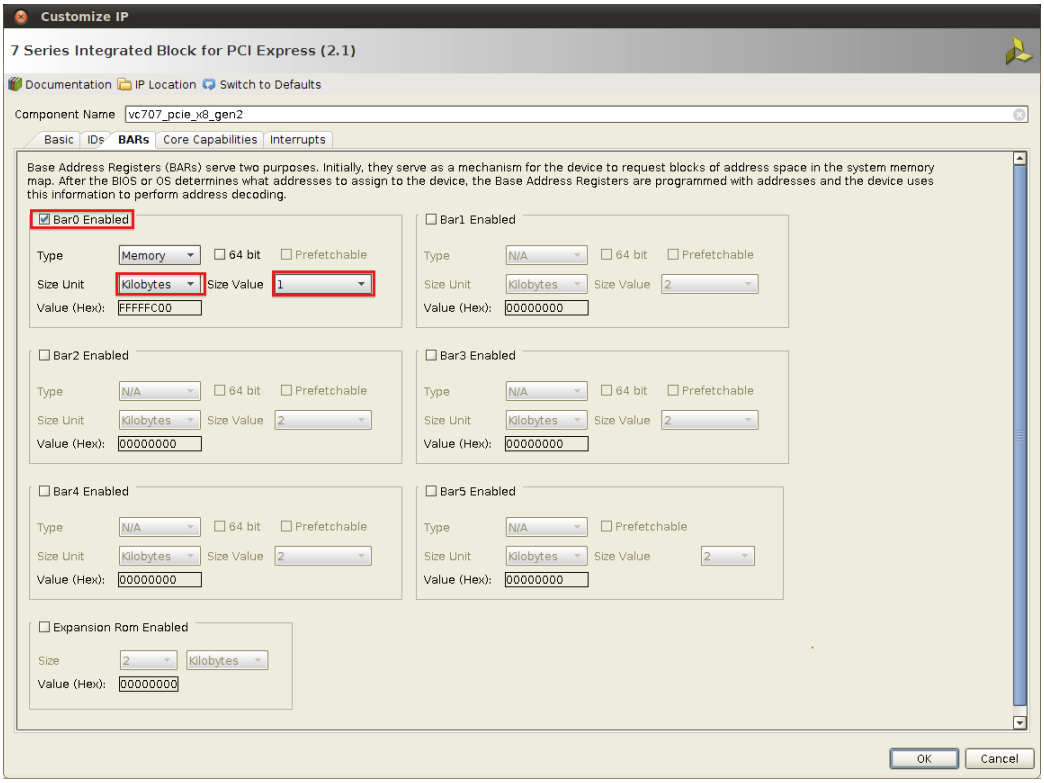


Figure B.21. Xilinx Vivado IP Catalog wizard screen.

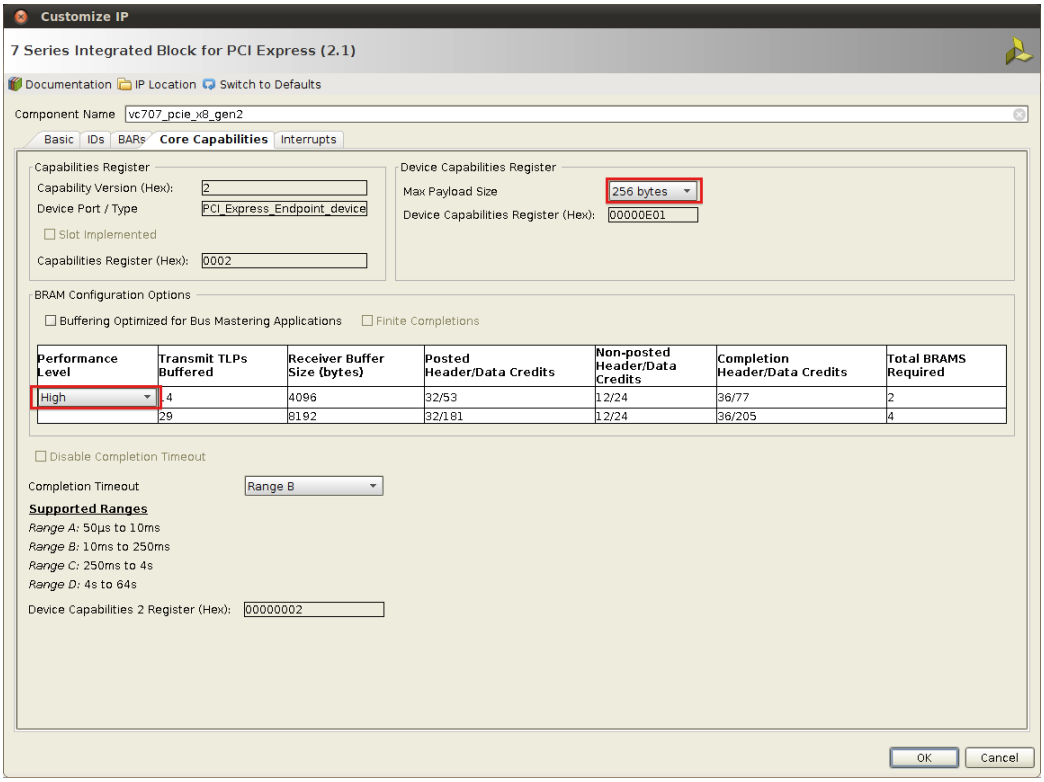


Figure B.22. Xilinx Vivado IP Catalog wizard screen.

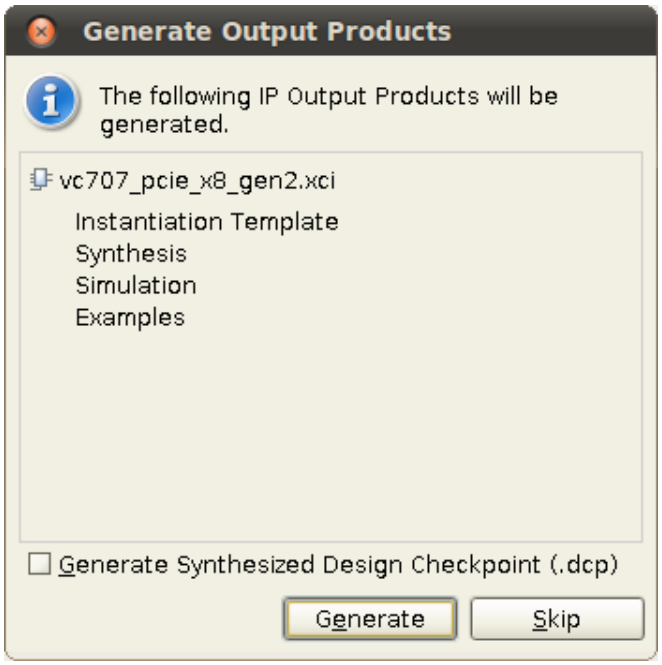


Figure B.23. Xilinx Vivado IP Catalog wizard dialog.

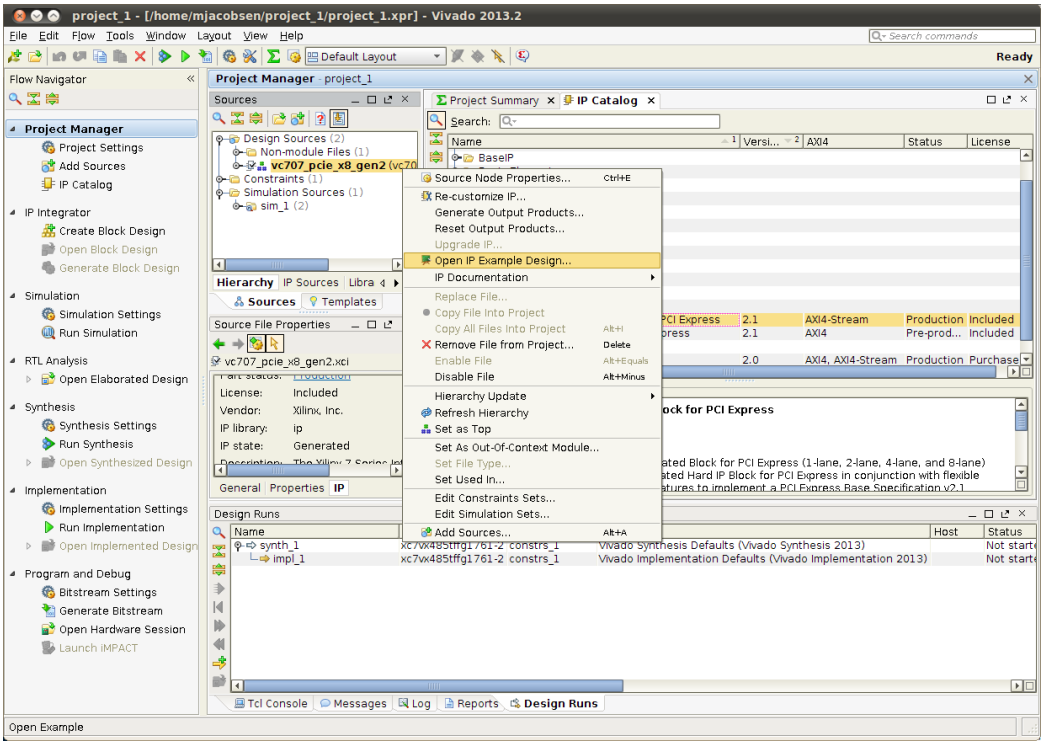


Figure B.24. Xilinx Vivado screen.

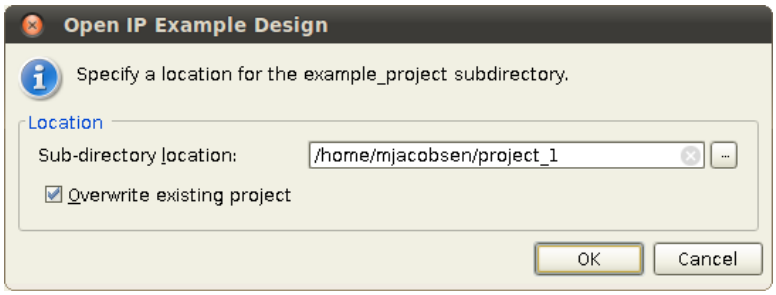


Figure B.25. Xilinx Vivado dialog.

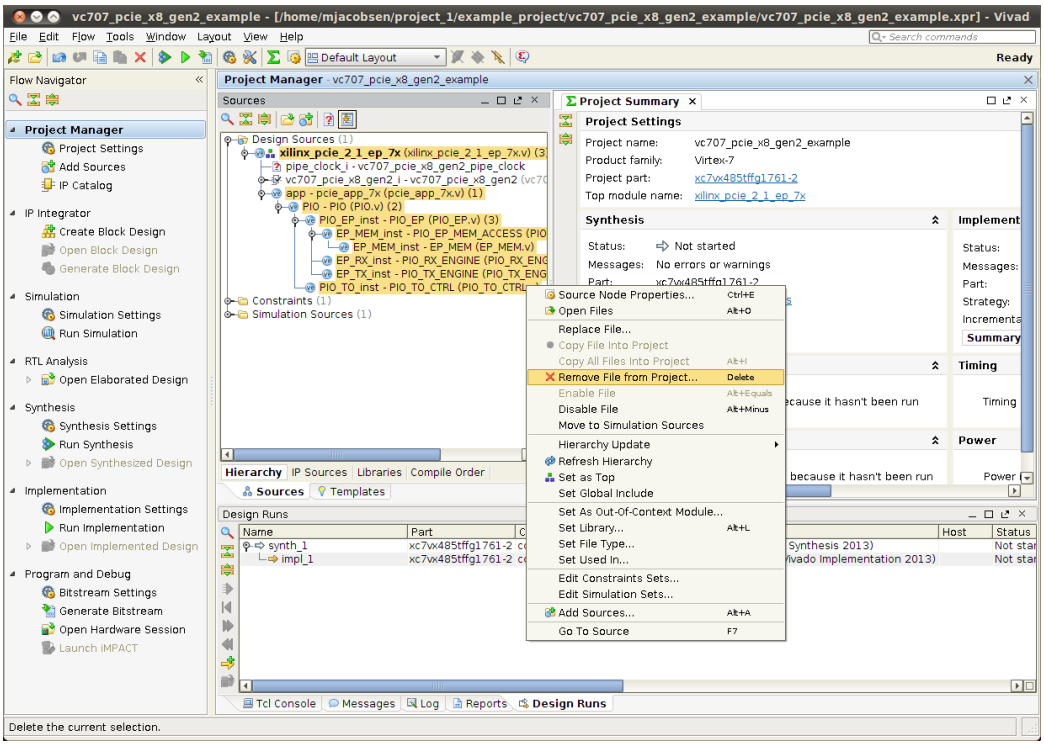


Figure B.26. Xilinx Vivado screen.

them from the project:

`xilinx_pcie_2_1_ep_7x.v`

`pcie_app_7x.v`

`PIO.v`

`PIO_EP.v`

`PIO_EP_MEM_ACCESS.v`

`EP_MEM.v`

`PIO_RX_ENGINE.v`

`PIO_TO_CTRL.v`

`PIO_TX_ENGINE.v`

This is pictured in Figure B.26.

Then add the RIFFA 2.0 HDL files to your project using the dialog pictured in

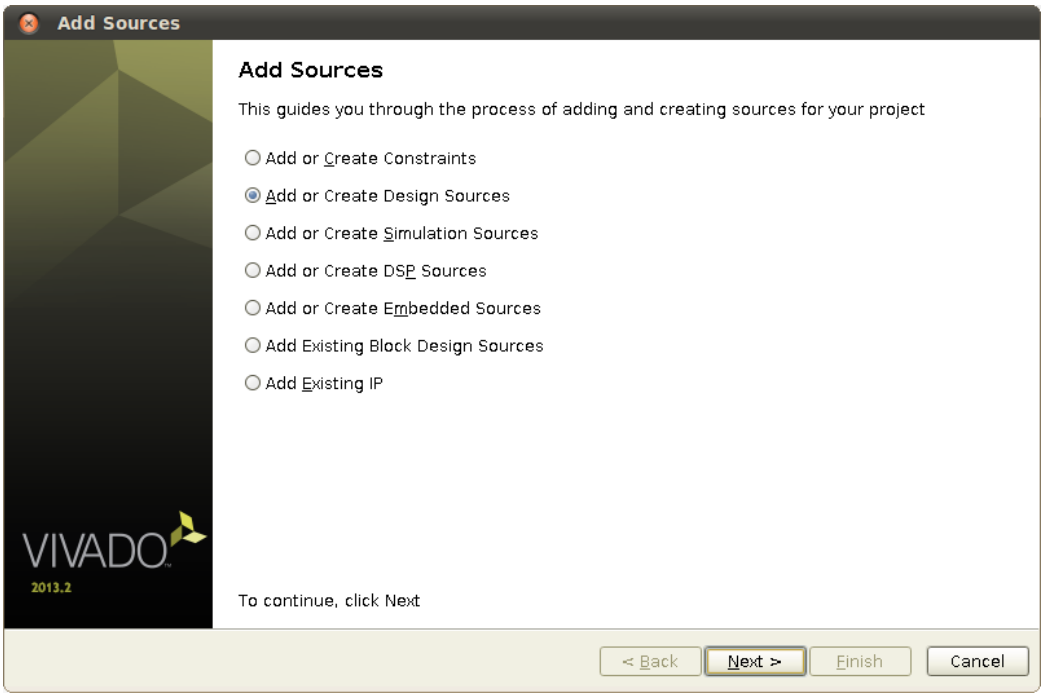


Figure B.27. Xilinx Vivado dialog.

Figure B.27.

Be sure to add the following files from the board directory:

`riffa_top_pcie_7x_v2_1.v`

`riffa_adapter_pcie_7x_v2_1.v`.

Once all the files are added, be sure Copy sources into project is checked and click Finish as pictured in Figure B.28.

4. Synthesize and implement. Before you can synthesize, you need to make modifications to the .xdc constraints file as per XTP207. This is pictured in Figure B.29. Add the following constraints:

```
set_property IOSTANDARD LVCMOS18 [get_ports emcclk]
set_property LOC AP37 [get_ports emcclk]
set_property BITSTREAM.CONFIG.BPI_SYNC_MODE Type1 [current_design]
set_property BITSTREAM.CONFIG.EXTMASTERCCLK_EN div-1 [current_design]
```

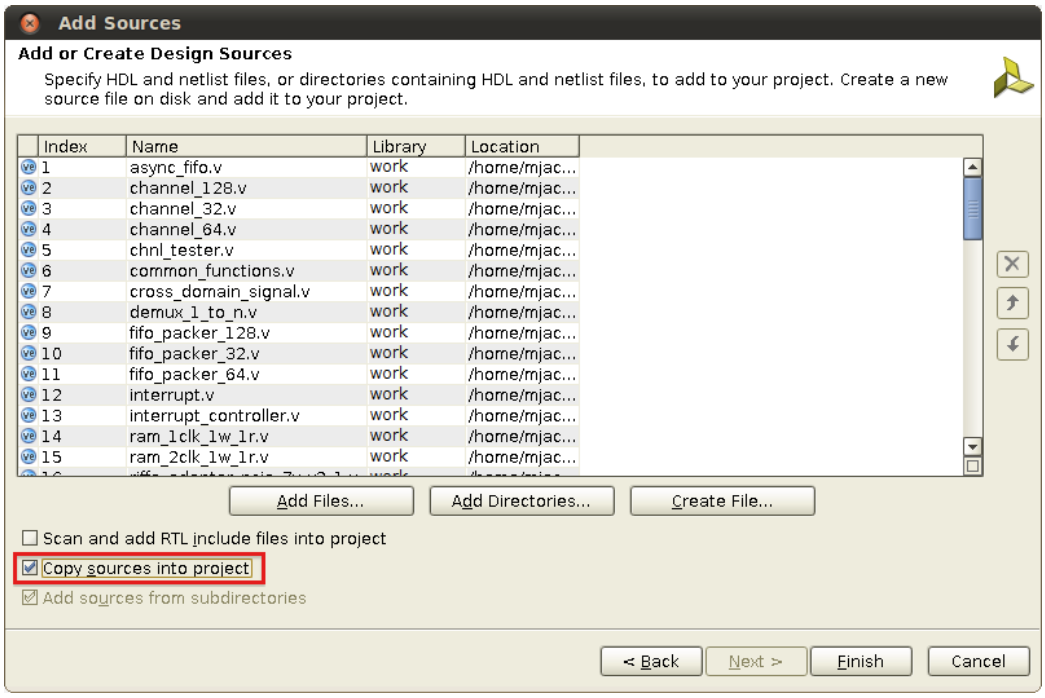


Figure B.28. Xilinx Vivado dialog.

```
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
```

At this point, if you attempt to synthesize you will encounter an error indicating C_NUM_CHNL is not defined. This is intentional. It is done to get you to open the RIFFA 2.0 adapter module file and edit it as needed. There are instructions in that file. However, all you need to do is uncomment the line that defines the C_NUM_CHNL parameter, set it to the number of channels you need (1-12), and you should be able to implement the design completely.

The adapter module instantiates a chnl_tester module for each channel. Sample user application software in the RIFFA 2.0 distribution can be used to send and receive data to/from the chnl_tester modules. The chnl_tester is meant to be an example. Your design will need to replace the chnl_tester modules with your own modules. You may also need to modify the .xdc, top level, and RIFFA adapter modules to bring in additional signals as dictated by your design.

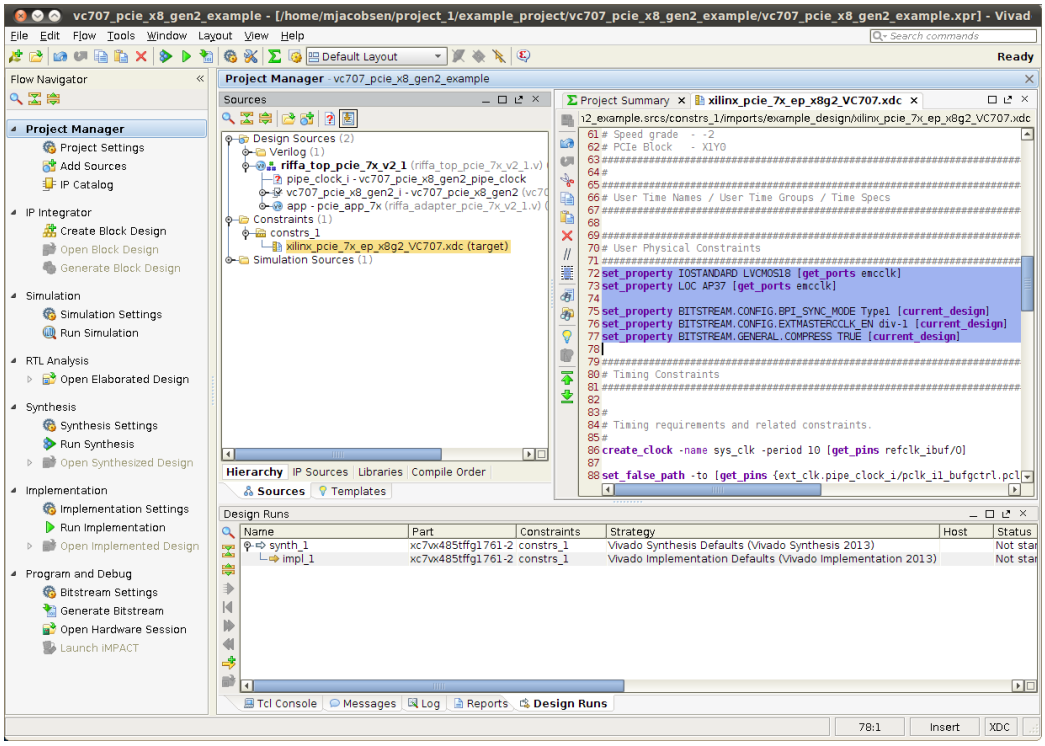


Figure B.29. Xilinx Vivado screen.

Appendix C

Reusable Components

The following is a listing and explanation of the reusable components in the Smart Frame Grabber Framework.

C.1 Image Scaler

A streaming image scaler module that supports arbitrary precision scaling. It provides support for bi-linear or nearest neighbor interpolation. This work was originally the design of David Kronstein and licensed by the GNU Lesser General Public License. However, it has been substantially reworked to support higher frequencies.

image_scaler.v Scales streaming video up or down in resolution. Bilinear and nearest neighbor modes are supported. Run-time adjustment of input and output resolution, scaling factors, and scale type. Pipeline delay of 7 cycles. Output begins after 2 lines of video have been acquired.

C.2 Frame Capture

Captures image frames at any resolution and rate. Makes the data available via a FIFO.

frame_capture.v Captures DVI video frame data and buffers a portion of it in a FIFO. Users should pulse the CAPTURE port high (using the RD_CLK) for each frame they want captured. The first complete video frame will be captured after each CAPTURE pulse. Pulses will not accrue. Only after a frame has completely rendered (i.e. during VS), will additional pulses have an effect.

Captured frame data is made available on RD_DATA via a FWFT asynchronous FIFO. RD_EMPTY will be high until valid data is available. The combination RD_EN and RD_EMPTY will consume the data on RD_DATA. EOF will be high for at least one cycle between frames after the last bit of frame data has been read out via RD_DATA. Note that requesting the next frame may need to happen before the current frame has been completely read out. VS will likely start before all the captured frame data has been read out of the buffer. So take care not to miss the entire VS period reading out data if continuous frame capture is needed.

If RD_EMPTY goes high before before an entire frame's worth of data has been provided, it indicates a truncated frame. This is likely the result of an overflow, or of an incomplete frame from the video source. Handle this as appropriate.

Also note that continuous capturing may result in the final frame being requested, but not read (i.e. at the end of a continuous capture sequence). Between continuous captures, users should assert RST for at least one cycle to clear out any buffered data from any previous capture sequence.

line_reverse.v Aggregates 24 bit data into a line (delimited by high DVI_HS) and output 24 bit data in the reverse order in which it was generated.

C.3 Sliding Window Framework

Provides the basic framework for sliding window style processing such as convolution or feature extraction.

sliding_window.v Saves image pixels in BRAM line buffers to support a vertical scrolling horizontal slice of image pixels across an image. The slice can be at most FLOOR(C_LINE_SIZE/C_SHARE_MULT) pixels wide and C_WIN_SIZE pixels high. The slice supports a left-right sliding window of size C_WIN_SIZE x C_WIN_SIZE. Once the end of a pixel line is reached, the window shifts down a line and begins scrolling left to right again. When at least C_WIN_SIZE rows and columns have been buffered, the sliding begins. On each new line, sliding is delayed until C_WIN_SIZE columns have been buffered.

The ADDR_A, ADDR_B, and LINE_OFF ports are used to request window data. When ADDR_VALID is high, these values must be valid. ADDR_RECVD will be pulsed high for every set of addresses read. The window values will be available on ports DATA_A and DATA_B 2 cycles later. DATA_VALID will also be high when these ports have valid values. After the last set of addresses are acknowledged (by the ADDR_RECVD pulse), ADDR_DONE should be pulsed high to signal the end of reading (for the current window).

NOTE: to achieve maximum throughput, some knowledge of the organization of data must be known by external modules. Window data is exposed through dual BRAM read ports. Two ports from _each_ BRAM can be read on every cycle. C_NUM_LINE BRAMs will store the window data. Each BRAM stores C_SHARE_MULT image lines. To access a pair of pixels on line n in the current window, use the (n MOD C_NUM_LINE)th port pair. The address to assert on ports ADDR_A and ADDR_B

is the position of the pixel in the window line, [0, C_WIN_SIZE-1]. A line offset is needed to distinguish which image line in the BRAM is the target. This value is FLOOR(n/C_NUM_LINE), in the range [0, C_SHARE_MULT-1]. It should be asserted on the LINE_OFF port. For example, if C_NUM_LINE == 10, to read positions 3 and 10 on line 13, use the (0,1,2,...) 3rd set of address and data ports. Then assert ADDR_A = 3, ADDR_B = 7, and LINE_OFF = 1. To access the same positions on line 3 in the window (instead of line 13), use the same ports and simply change LINE_OFF to 0.

Input data is provided on the PX_EOF, PX_EOL, PX_DE, and PX_DATA ports. When PX_VALID is high, their values are valid. PX_VALID must remain high until PX_RECVD is pulsed high to acknowledge the receipt. This also requests the next pixel. PX_EOL must be pulsed high for 1 cycle after every line. PX_EOF must be pulsed high for 1 cycle after all the lines (at the end of a frame). When PX_DE is high, that indicates PX_DATA contains pixel information. PX_DE must not be high when either PX_EOL or PX_EOF are high.

C.4 Integral Image Conversion

Conversion support for integral images.

integral_image.v Calculates integral image pixels from C_PX_WIDTH bit grayscale pixels. Outputs a C_II_WIDTH bit integer after a 4 cycle delay (as indicated by DE_OUT). HS_IN should go high for least one cycle between lines. VS_IN should go high for at least one cycle between frames.

C.5 Pixel Color Space Conversion

Color space conversion utilities.

rgb_to_gray.v Converts 24 bit RGB video to grayscale 8 bit video with a 3 cycle delay.

rgb_to_hsv.v Converts the input RGB data into output HSV data. The supplied address follows the input pixel to output pixel.

C.6 Arithmetic Operations

Parameterized addition utilities to automatically build a pipelined adder tree. This allows a function that sums multiple value per cycle to be spread out over many cycles.

binary_adder_tree.v Recursively builds a binary adder tree and returns the SUM. Each level in tree is registered. The VALID_OUT signal is a delayed version of the VALID_IN and corresponds to the calculation latency. Cycle delay will be clog2(C_NUM_OPS).

binary_signed_adder_tree.v Recursively builds a signed binary adder tree and returns the SUM. Each level in tree is registered. The VALID_OUT signal is a delayed version of the VALID_IN and corresponds to the calculation latency. Cycle delay will be clog2(C_NUM_OPS).

calc_mean.v Calculates the mean and squared mean of the specified values. The values are added to a running sum when VALID_IN is high. When VAL_DONE is high, the sums are divided by the count and the MEAN and SQ_MEAN are outputted with VALID_OUT high. RST must be asserted high between calculations. MEAN, and SQ_MEAN are 32 bit IEEE 754 floats.

Note that to work properly the following must be true:
CYCLES_BETWEEN_VALID_IN_PULSES ¿ C_ADD_SUB_DELAY + 3

C.7 Counting and Filtering

Counting and region identification routines for windows and pixels.

annular_filter.v Creates an annular (ring) region based on the radii provided and samples according to the sampling frequency within the annular region. The region is the area ¡ RADIUS_SQ_OUTR and ≥ RADIUS_SQ_INNR, as measured by the RADIUS_SQ port.

The C_MAX_SAMPLE_FREQ should be set to a number that is the maximum count between valid windows during which no sampling will take place. For example, if C_MAX_SAMPLE_FREQ == 256, then the sampling density must be no less than 1 every 256 valid windows (or 0.00390625). SAMPLE_FREQ is the actual sampling frequency.

pixel_counter.v Counts pixels to output X,Y coordinates with the video data. Coordinates start at zero and go to width - 1, and height -1.

C.8 Feature Extraction and Calculation

Cores that work with the sliding window framework to calculate features.

ncc_parms_and_calc.v Stores image data and template data to calculate normalized cross-correlation values on the image. The C_WIN_DIM x C_WIN_DIM template pixels are cross-correlated with each C_WIN_DIM x C_WIN_DIM image window. The calculation is parallelized by a factor of C_WIN_DIM to provide a new value every C_WIN_DIM cycles.

Template data is stored in a C_WIN_DIM x C_PIXEL_WIDTH bit wide BRAM, C_WIN_DIM values deep. It is received one C_WIN_DIM x C_PIXEL_WIDTH bit

column per cycle. Image data is stored in a similar sized BRAM and received in a similar way, one column per cycle. However, image data is stored in a circular fashion in the BRAM.

haar_calc.v Calculates Haar feature values using the specified coordinates and weights on the specified window data (in WIN). C_NUM_RECTS rectangles will be calculated in parallel on the window. The result is available on the VALUE port when VALID_OUT is high. Setting C_NUM_REG_OUT_STAGES to something other than 0 will register the outputs of each rectangle sum. This will add delay and increase register usage, but can improve timing and fmax by only adding 1 or 2 stages. C_MAX_COORD represents the maximum coordinate within the window that will be accessed. C_MAX_COORD must be less than or equal to C_WIN_DIM (if the coordinates are a subset, the mux's will be fewer).

The WIN window data must be organized in rows of pixels such that the top row starts at address 0. Within each row, the pixels must be arranged so the newest pixel in that row is on the left. For example, a 3x3 window at the upper left corner of the image with pixel widths of 4 will look like:

WIN[11:00] = {PX(2,0), PX(1,0), PX(0,0)}

WIN[23:12] = {PX(2,1), PX(1,1), PX(0,1)}

WIN[35:24] = {PX(2,2), PX(1,2), PX(0,2)}

Here PX(x,y) corresponds to the pixel at coordinate x, y in the image. See sliding_window.v for further details.

C.9 Clock Domain Crossing

Utilities to help avoid metastability issues when crossing between clock domains.

syncff.v A back to back FF design to mitigate the metastable issues when dealing with signaling between different frequency clocks (or asynchronous signals).

C.10 Data Manipulation

Data aggregation, distribution, and manipulation utilities.

pack_24_to_32.v Aggregates 24 bit data and outputs 32 bit word data. This is used to pack data into a format easy to send via multiples of 32 bits.

distribute_bytes.v Distributes bytes, C_OUT_BYTES at a time from a small cache. Reads in C_IN_BYTES bytes at a time from a connected FWFT FIFO. C_IN_BYTES must be an integer multiple of C_OUT_BYTES and C_IN_BYTES $\geq$ C_OUT_BYTES. When FLUSH pulses high, this module will pulse FLUSHED after all the remaining bytes in the FWFT FIFO are read and distributed (that is, as soon as INDATA_EN goes low after the FLUSH pulse). The module will not wait for more data to be made available in the FWFT FIFO.

accumulate_bytes.v Accumulates bytes, C_IN_BYTES at a time until C_OUT_BYTES bytes have been collected. At which point, EMPTY will drop. NOTE: C_OUT_BYTES MUST BE AN INTEGER MULTIPLE OF C_IN_BYTES AND C_OUT_BYTES $\geq$ C_IN_BYTES. When FLUSH is high, any data collected but not yet outputted will be outputted on

C.11 RAMs and FIFOs

These modules infer RAM blocks and FIFO queues during FPGA synthesis.

ram_1clk_1rw.v An inferrable RAM module. Single clock, 1 read/write port. For Xilinx, specify RAM_STYLE="BLOCK" or "DISTRIBUTED" to use BRAM or LUT memory.

Specify the width and depth as parameters.

ram_1clk_1rw_1r.v An inferrable RAM module. Single clock, 1 read/write port, 1 read port. For Xilinx, specify RAM_STYLE="BLOCK" or "DISTRIBUTED" to use BRAM or LUT memory.

Specify the width and depth as parameters.

ram_1clk_1w_1r.v An inferrable RAM module. Single clock, 1 write port, 1 read port. For Xilinx, specify RAM_STYLE="BLOCK" or "DISTRIBUTED" to use BRAM or LUT memory.

Specify the width and depth as parameters.

ram_1clk_2rw.v An inferrable RAM module. Single clock, 2 read/write ports. For Xilinx, specify RAM_STYLE="BLOCK" or "DISTRIBUTED" to use BRAM or LUT memory.

Specify the width and depth as parameters.

ram_2clk_1w_1r.v An inferrable RAM module. Dual clocks, 1 write port, 1 read port. For Xilinx, specify RAM_STYLE="BLOCK" or "DISTRIBUTED" to use BRAM or LUT memory.

Specify the width and depth as parameters.

ram_2clk_2rw.v An inferrable RAM module. Dual clocks, 2 read/write ports. For Xilinx, specify RAM_STYLE="BLOCK" or "DISTRIBUTED" to use BRAM or LUT

memory.

Specify the width and depth as parameters.

sync_fifo.v A synchronous capable parameterized FIFO. With a traditional FIFOs, the RD_DATA will be valid one cycle following a RD_EN assertion. EMPTY will remain low until the cycle following the last RD_EN assertion. Note, that EMPTY may actually be high on the same cycle that RD_DATA contains valid data.

Specify the width and depth as parameters.

sync_fifo_fwft.v A synchronous capable parameterized first word fall through FIFO. As with all first word fall through FIFOs, the RD_DATA will be valid when RD_EMPTY is low. Asserting RD_EN will consume the current RD_DATA value and cause the next value (if it exists) to appear on RD_DATA on the following cycle. Be sure to check if RD_EMPTY is low each cycle to determine if RD_DATA is valid.

Specify the width and depth as parameters.

async_fifo.v An asynchronous capable parameterized FIFO. With a traditional FIFOs, the RD_DATA will be valid one cycle following a RD_EN assertion. EMPTY will remain low until the cycle following the last RD_EN assertion. Note, that EMPTY may actually be high on the same cycle that RD_DATA contains valid data.

Specify the width and depth as parameters.

async_fifo_fwft.v An asynchronous capable parameterized first word fall through FIFO. As with all first word fall through FIFOs, the RD_DATA will be valid when RD_EMPTY is low. Asserting RD_EN will consume the current RD_DATA value and cause the next value (if it exists) to appear on RD_DATA on the following cycle. Be sure to check if RD_EMPTY is low each cycle to determine if RD_DATA is valid.

Specify the width and depth as parameters.

C.12 Skin Detector

A HSV based pixel classifier that uses a boost trained alternating decision tree for evaluation.

SDHSVBuilder.java When run with a suitable alternating decision tree .xml file as input, generates a skin_detector_hsv.v Verilog module that evaluates each pixel in a pipelined fashion to classify. Parameterized to support the number of classification operations to calculate per pipeline stage.

Bibliography

[1] Amit Adam, Ehud Rivlin, and Ilan Shimshoni. Robust fragments-based tracking using the integral histogram. In *CVPR*, 2006.

[2] Andrieu, de Freitas, Doucet, and Jordan. An introduction to MCMC for machine learning. *MACHLEARN: Machine Learning*, 50, 2003.

[3] Miguel Arias-Estrada and Eduardo Rodríguez-Palacios. An FPGA co-processor for real-time visual tracking. *Lecture Notes in Computer Science*, 2438:710, 2002.

[4] Miguel Arias-Estrada and Eduardo Rodríguez-Palacios. An fpga co-processor for real-time visual tracking. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02, pages 710–719, London, UK, 2002. Springer-Verlag.

[5] Akshay Athalye, Miodrag Bolic, Sangjin Hong, and Petar M. Djuric. Generic hardware architectures for sampling and resampling in particle filters. *EURASIP J. Adv. Sig. Proc*, 2005(17):2888–2902, 2005.

[6] Shai Avidan. Support vector tracking. *IEEE Trans. Pattern Anal. Mach. Intell*, 26(8):1064–1072, 2004.

[7] Boris Babenko and Ming-Hsuan Yang Serge Belongie. Robust object tracking with online multiple instance learning. *TPAMI*, 2011.

[8] Sebastian Bauer, Sebastian Kohler, Konrad Doll, and Ulrich Brunsmann. Fpga-gpu architecture for kernel svm pedestrian detection. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 61–68. IEEE, 2010.

[9] Gary R. Bradski. Computer vision face tracking for use in a perceptual user interface. *Intel Technology Journal*, 1998.

[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] R. J. Butera, C. G. Wilson, C. A. Delnegro, and J. C. Smith. A methodology for achieving high-speed rates for artificial conductance injection in electrically excitable biological cells. *IEEE Trans Biomed Eng*, 48(12):1460–1470, Dec 2001.

[12] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Proc. ACM Conf. Human Factors in Computing Systems, CHI*, pages 181–188. ACM, April 1991.

[13] Jeff Chase, Brent Nelson, John Bodily, Zhaoyi Wei, and Dah-Jye Lee. Real-time optical flow calculations on fpga and gpu architectures: A comparison study. In *FCCM*, pages 173–182, 2008.

[14] Kamalika Chaudhuri, Yoav Freund, and Daniel Hsu. Tracking using explanation-based modeling. 2009.

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

[16] Jung Uk Cho, Seung Hun Jin, Xuan Dai Pham, Jae Wook Jeon, Jong Eun Byun, and Hoon Kang. A real-time object tracking system using a particle filter. In *Intelligent Robots and Systems*, 2006.

[17] Jung Uk Cho, Seunghun Jin, Xuan Dai Pham, and Jae Wook Jeon. Multiple objects tracking circuit using particle filters with multiple features. In *ICRA*, pages 4639–4644. IEEE, 2007.

[18] Junguk Cho, Shahnam Mirzaei, Jason Oberg, and Ryan Kastner. Fpga-based face detection system using haar classifiers. In Paul Chow and Peter Y. K. Cheung, editors, *FPGA*, pages 103–112. ACM, 2009.

[19] Junguk Cho, Shahnam Mirzaei, Jason Oberg, and Ryan Kastner. Fpga-based face detection system using haar classifiers. In *FPGA*, 2009.

[20] Adam Coates, Paul Baumstarck, Quoc V. Le, and Andrew Y. Ng. Scalable learning for object detection with GPU hardware. In *IROS*, pages 4287–4293. IEEE, 2009.

[21] Ben Cope, Peter Y. K. Cheung, Wayne Luk, and Sarah Witt. Have GPUs made FPGAs redundant in the field of video processing? In *FPT*, pages 111–118. IEEE, 2005.

[22] T. G Dietterich, R. H. Lathrop, and T. Lozano-Perez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.

[23] A. D. Dorval, D. J. Christini, and J. A. White. Real-Time linux dynamic clamp: a fast and flexible way to construct virtual ion channels in living cells. *Ann Biomed Eng*, 29(10):897–907, Oct 2001.

[24] B. Echebarria and A. Karma. Spatiotemporal control of cardiac alternans. *Chaos*, 12(3):923–930, Sep 2002.

[25] Ken Eguro. SIRC: An extensible reconfigurable computing communication API. In Ron Sass and Russell Tessier, editors, *FCCM*, pages 135–138. IEEE Computer Society, 2010.

[26] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *FPGA*, pages 47–56. ACM, 2012.

[27] Yoav Freund, Robert Schapire, and N Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[28] Alex Goldhammer and John Ayer Jr. Understanding performance of pci express systems. *White Paper: Xilinx Virtex-4 and Virtex-5 FPGAs*, 2008.

[29] Naga K. Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. Memory - A memory model for scientific algorithms on graphics processors. In *SC*, page 89. ACM Press, 2006.

[30] H. Grabner, M. Grabner, and H. Bischof. Real-time tracking via on-line boosting. In *BMVC*, page I:47, 2006.

[31] Helmut Grabner, Michael Grabner, and Horst Bischof. Real-time tracking via on-line boosting. In *BMVC*, volume 1, page 6, 2006.

[32] Helmut Grabner, Christian Leistner, and Horst Bischof. Semi-supervised on-line boosting for robust tracking. In *Computer Vision–ECCV*. 2008.

[33] G. M. Hall and D. J. Gauthier. Experimental control of cardiac muscle alternans. *Phys. Rev. Lett.*, 88(19):198102, May 2002.

[34] Simon Heinzle, Pierre Greisen, David Gallup, Christine Chen, Daniel Saner, Aljoscha Smolic, Andreas Burg, Wojciech Matusik, and Markus H. Gross. Computational stereo camera system with programmable control loop. *ACM Trans. Graph*, 30(4):94, 2011.

[35] John M III. Open component portability infrastructure (opencpi), 2009.

[36] Ra Inta, David John Bowman, and Susan M. Scott. The "chimera": An off-the-shelf CPU/GPGPU/FPGA hybrid computing platform. *Int. J. Reconfig. Comp*, 2012.

[37] S. Iravanian and D. J. Christini. Optical mapping system with real-time control capability. *Am. J. Physiol. Heart Circ. Physiol.*, 293(4):H2605–2611, Oct 2007.

[38] Matthew Jacobsen, Yoav Freund, and Ryan Kastner. Riffa: A reusable integration framework for fpga accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2012*, pages 216–219. IEEE, 2012.

[39] Matthew Jacobsen and Ryan Kastner. Riffa 2.0: A reusable integration framework for fpga accelerators. In *FPL*, 2013.

[40] Matthew Jacobsen, Pingfan Meng, Siddarth Sampangi, and Ryan Kastner. Fpga accelerated online boosting for multi-target tracking. In *Proceedings of the 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2014.

[41] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *PAMI*, 2012.

[42] 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*, pages 9–15, June.

[43] John H. Kelm and Steven S. Lumetta. Hybridos: runtime support for reconfigurable accelerators. In *FPGA*, pages 212–221, New York, NY, USA, 2008. ACM.

[44] H. C. Lai, M. Savvides, and T. H. Chen. Proposed FPGA hardware architecture for high frame rate (> 100 fps) face detection using feature cascade classifiers. In *Biometrics: Theory, Applications, and Systems*, pages 1–6, 2007.

[45] Charles Lo and Paul Chow. A high-performance architecture for training viola-jones object detectors. In *FPT*. IEEE, 2012.

[46] Pingfan Meng, Ali Irturk, Ryan Kastner, Andrew McCulloch, Jeffrey Omens, and Adam Wright. Gpu acceleration of optical mapping algorithm for cardiac electrophysiology. In *EMBC*, Aug 2012.

[47] David Merrill. Head-tracking for gestural and continuous control of parameterized audio effects. In *Proceedings of the 2003 conference on New interfaces for musical expression*, pages 218–219. National University of Singapore, 2003.

[48] R. B. Miller. Response time in man-computer conversational transactions. *FALL JOINT COMPUTER CONF.*, pages 267–277, 1968.

[49] H. N. Pak, Y. B. Liu, H. Hayashi, Y. Okuyama, P. S. Chen, and S. F. Lin. Synchronization of ventricular fibrillation with real-time feedback pacing: implication to low-energy defibrillation. *Am. J. Physiol. Heart Circ. Physiol.*, 285(6):H2704–2711, Dec 2003.

[50] Karl Pauwels, Matteo Tomasi, Javier Diaz, Eduardo Ros, and Marc M. Van Hulle. A comparison of fpga and gpu for real-time phase-based optical flow, stereo, and local image features. *IEEE Transactions on Computers*, 61:999–1012, 2012.

[51] Wesley Peck, Erik K. Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David L. Andrews. Hthreads: A computational model for reconfigurable devices. In *FPL*, pages 1–4. IEEE, 2006.

[52] Karl Pereira, Peter Athanas, Heshan Lin, and Wu Feng. Spectral method characterization on FPGA and GPU accelerators. In *ReConFig*, pages 487–492. IEEE Computer Society, 2011.

[53] David A Ross, Jongwoo Lim, Ruei-Sung Lin, and Ming-Hsuan Yang. Incremental learning for robust visual tracking. *IJCV*, 2008.

[54] D. Snow, M. J. Jones, and P. Viola. Detecting pedestrians using patterns of motion and appearance. In *ICCV*, 2003.

[55] 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*, 23(3), 2009.

[56] D. Sung, J. Somayajula-Jagai, P. Cosman, R. Mills, and A. D. McCulloch. Phase shifting prior to spatial filtering enhances optical recordings of cardiac action potential propagation. *Ann Biomed Eng*, 29(10):854–61, 2001.

[57] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with fpgas and gpus. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124. ACM, 2010.

[58] Paul A. Viola and Michael J. Jones. Robust real-time face detection. In *ICCV*, page 747, 2001.

[59] Paul A. Viola, John C. Platt, and Cha Zhang. Multiple instance boosting for object detection. In *NIPS*, 2005.

[60] J. Y. Wang, X. L. Chen, and W. Gao. Online selecting discriminative tracking features using particle filter. In *CVPR*, pages II: 1037–1042, 2005.

[61] Y. Wei, X. Bing, and C. Chareonsak. Fpga implementation of adaboost algorithm for detection of face biometrics. In *Biomedical Circuits and Systems, 2004 IEEE International Workshop on*, pages S1/6 – 17–20, 2004.