Build Your Own Domain-specific Solutions with RapidWright

Chris Lavin and Alireza Kaviani
Xilinx Research Labs
2/24/19
Why are Domain-specific solutions important?

- RapidWright value proposition
- Why open source?

What is RapidWright?

How to use RapidWright?
FPGA Industry and Community Dynamics

- Continuous industry and community engagement
The Age of Domain Specific Architectures

- Achieve higher efficiency by tailoring the architecture to characteristics of the domain
  - More effective parallelism for a specific domain, More effective use of memory bandwidth
  - Domain specific programming language

Source: A New Golden Age for Computer Architecture: (Domain-Specific Hardware/Software Co-Design)
John Hennessy and David Patterson
Stanford and UC Berkeley, 13 June 2018
Raising the Abstraction of Design Entry

FPGA Tool/Interface Stack

- Overlays
- SDx (SDAccel, SDSoC)
- HLS
- IPI
- RTL Synthesis

Place & Route

- Device models, Bitstream
- Device, Heterogenous Architecture components

FPGA Silicon
- Devices + P&R
- 3rd party HDL synthesis

FPGA compilers
- HLS + Open Source

Platform compilers
- SDx, OpenCL

DSA compilers
- Domain expertise + open source

Highest Productivity

Highest Performance

> Enabling domain specific performance optimizations

© Copyright 2019 Xilinx
RapidWright Value Proposition

IMPLEMENTATION COMPILE TIME

PERFORMANCE

FPGAs + Pre-implemented blocks

CPUs

FPGAs + Shells

RapidWright

FPGAs

ASICs

SDx
Focus on Emerging Applications

- Module-based approach to implementation
  - Lock-in performance with reusable modules
  - Fewer inter-block timing closure issues

- Designs with Replication

- Latency-flexible Connectivity

- Goals
  - Productivity
    - Order of magnitude reduction in compile time per domain
  - Performance (near-spec)
  - Predictable timing closure
Proposed Domain-specific Tool Flows

Developers Driving Design
- Data scientists, application architects
- Academic and industrial community
- Xilinx and open source community

Tools and Frameworks
- High abstraction Domain-specific language
- Domain-specific data flow graph (LLVM) compiler

Front-end Compiler
- Relocate pre-implemented operators and functions

Back-end compiler

Application in Domain 1
Application in Domain 2
Application in Domain 3

VIVADO

© Copyright 2019 Xilinx
> **Fact**
  >> Emerging domains such as surveillance or vision have high replication

> **Community role**
  >> Identify and extract operators and functions in the domain

> **RapidWright value proposition**
  >> Assemble relocatable pre-implemented domain operators
  >> Deliver the best inference/watt
Building Relocatable Domain-specific Shells

- Fact
  - Advances in silicon have created QoR opportunity

- Community role
  - Domain-specific shell design or overlays

- RapidWright value proposition
  - Achieve near-spec performance

Series 7 (28ns)  UltraScale (20ns)  UltraScale+ (16nm)

- Logic & DSP
- BRAM
- Out of the box Fmax

The QoR Opportunity
Success Scenario: Rapid Domain-specific Assembly

<table>
<thead>
<tr>
<th></th>
<th>RAPID WRIGHT</th>
<th>VIVADO</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation Time</td>
<td>~ 6 mins</td>
<td>~1-3 Hours</td>
<td>10-30X</td>
</tr>
<tr>
<td>Accelerator Fmax</td>
<td>~700MHz</td>
<td>~450MHz</td>
<td>~1.5X</td>
</tr>
</tbody>
</table>
What is RapidWright?
RapidWright Overview

> **Companion framework for Vivado**
  > Fast, light-weight, open source
  > Communicates through Design CheckPoints\(^1\) (DCPs)
  > Java code, Python scripting

> **Enables targeted solutions**
  > Reuse & relocate pre-implemented modules
  > Just-in-time implementations
  > Create shells & overlays

> **Power user ecosystem**
  > Academic algorithm validation
  > Rapid prototyping of CAD concepts

\(^1\)DCP = netlist + P&R data + constraints
4 Ways to Design in RapidWright

**BUILD ROUTED CIRCUITS**
- FROM SCRATCH
- Well-defined circuits in seconds
- Parameterizable library of generators

**GENERATORS**
- FROM VIVADO
- REUSE P&R CIRCUITS
- FROM SHELLS & OVERLAYS
- Reuse/relocate P&R circuits from Vivado
- Combine P&R circuits together
A Modular Pre-implemented Methodology

USER TASKS (MANUAL)
1. Design selection attributes:
   - Modular
   - Latency tolerant
   - Prefers replication
2. Placement planning

TOOL TASKS (AUTOMATED)
3. P&R modules cached:
   - Relocatable
   - Reusable
   - Timing predictable
4. Run implementation

Match Design Structure to Architecture Patterns

© Copyright 2019 Xilinx
Creating Pre-implemented Modules (Vivado OOC Flow)

OOC Synthesis

Area Constraint

OOC Place & Route

Block Packager

Block Cache
RapidWright Pre-implemented Module Flow

- User Design
- Design Parser
- Block Assembler
- Block Cache
- Block Placer
- Route Design
- Final Impl.

Fully Placed, Partially Routed Implementation
## Design Performance Results

<table>
<thead>
<tr>
<th>Design</th>
<th>Target Device</th>
<th>Baseline (initial design)</th>
<th>RapidWright¹ Flow</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seismic</td>
<td>KU040</td>
<td>270MHz</td>
<td>390MHz</td>
<td>41%</td>
</tr>
<tr>
<td>FMA</td>
<td>KU115</td>
<td>270MHz</td>
<td>417MHz</td>
<td>54%</td>
</tr>
<tr>
<td>GEMM</td>
<td>KU115</td>
<td>391MHz</td>
<td>462MHz</td>
<td>16%</td>
</tr>
<tr>
<td>ML overlay</td>
<td>ZU9EG</td>
<td>368MHz</td>
<td>541MHz</td>
<td>50%</td>
</tr>
</tbody>
</table>

### Utilization table

<table>
<thead>
<tr>
<th>Design</th>
<th>LUT</th>
<th>FF</th>
<th>DSP</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seismic</td>
<td>93%</td>
<td>5%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FMA (HPC design)</td>
<td>25%</td>
<td>50%</td>
<td>97%</td>
<td>6%</td>
</tr>
<tr>
<td>GEMM</td>
<td>19%</td>
<td>20%</td>
<td>87%</td>
<td>-</td>
</tr>
<tr>
<td>ML overlay</td>
<td>46%</td>
<td>29%</td>
<td>42%</td>
<td>96%</td>
</tr>
</tbody>
</table>

¹RapidWright: Enabling Custom Crafted Implementations for FPGAs, FCCM 2018
Re-locatability & Reuse of Multiple Implementations

<table>
<thead>
<tr>
<th>RUN</th>
<th>$F_{\text{MAX}}$ (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vivado</td>
<td>270</td>
</tr>
<tr>
<td>RapidWright</td>
<td>417 (+53%)</td>
</tr>
</tbody>
</table>

- > 97% DSP utilization
- > 4.4 TeraOp/s
- > “Fabric discontinuites”
  - >> SLR boundary
  - >> IO Columns
  - >> Laguna Tiles

Impl #0  Impl #2  Impl #4
Impl #1  Impl #3  Impl #5

© Copyright 2019 Xilinx
Latency Flexibility: AXI Stream Register Slices

> Exploiting latency-tolerance and architectural knowledge
  >> Automatic insertion of latency blocks
Debugging with an ILA (ChipScope)

I downloaded my design and it’s not working. But it works in simulation!

I added an ILA, but the bug is gone!

You’ll need to recompile with an ILA to debug it.
Experiment: Insert Pre-implemented ILA

> Preserves existing
  >> Placement
  >> Routing

> Only occupy unused resources
Preserve Existing Placement & Routing

Debug Blocks Inserted by RapidWright
Debug Instrumentation Speedup

- Baseline
- RapidWright Debug

Runtime (minutes)

- dsp1 (9% CLBs): 35x
- 10g (10% CLBs): 24x
- dsp2 (20% CLBs): 12x
- sparc (31% CLBs): 97x
- 21ch (70% CLBs): 33x
Beyond a Pre-implemented Methodology

RapidWright probe router enables higher productivity

- 21X more debug turns per day
- Highest level of routing preservation possible
- Future innovation:
  - iteration with extra probe inputs
  - Automatic insertion of pipeline flops to manage timing

<table>
<thead>
<tr>
<th>Vivado</th>
<th>RapidWright</th>
<th>Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td>modify_debug_probes</td>
<td>ProbeRouter</td>
<td>21X</td>
</tr>
<tr>
<td>130 mins</td>
<td>6.3 mins</td>
<td></td>
</tr>
</tbody>
</table>

Original

RapidWright Probes Rerouted
Pre-implemented Data Movement Shell

> Goals
  >> Minimize overhead of compute (and overlays)
  >> Prove shell assembly model

> Build-to-order LinkBlaze\(^1\) shell
  >> 512 bit, bi-directional
  >> RapidWright Pre-implemented modules

<table>
<thead>
<tr>
<th>Vivado</th>
<th>RapidWright</th>
</tr>
</thead>
<tbody>
<tr>
<td>516MHz</td>
<td>620MHz (+20%)</td>
</tr>
</tbody>
</table>

\(^1\) LinkBlaze: Efficient global data movement for FPGAs (ReConFig 2017)
Just-in-time, Circuit Module Generators

> **Build modules on-demand**
  >> Placed and routed *in seconds*
  >> Reusable and compose-able
  >> Target spec performance

> **Parameterizable Generators**
  >> Adder
  >> Subtractor
  >> Multiplier

> **Expression Generator**
  >> Invokes math generators
  >> Built to spec: 775MHz

```
x^2 + 3*x - 5
```
RapidWright SLR Crossing DCP Creator

SLR crossing module from scratch

- Parameterizable
- Closes timing at 760MHz
  - Clk Period: 1.313ns
- Routed clock, placed and routed
- Runs in seconds

---

This RapidWright program creates a placed and routed DCP that can be imported into UltraScale+ designs to aid in high speed SLR crossings. See RapidWright documentation for more information.

Option                                   Description
----- -------------------------------------
-?, -h                                   Print Help
-a [String: Clk input net name]          (default: clk_in)
-b [String: Clock BUFGCE site name]      (default: BUFGCE_X0Y218)
-c [String: Clk net name]                (default: clk)
-d [String: Design Name]                 (default: slr_crosser)
-i [String: Input bus name prefix]       (default: input)
-l [String: Comma separated list of Laguna sites for each SLR crossing]
-n [String: North bus name suffix]       (default: _north)
-o [String: Output DCP File Name]        (default: slr_crosser.dcp)
-p [String: UltraScale+ Part Name]       (default: xcvu9p-flgc2104-2-i)
-q [String: Output bus name prefix]      (default: output)
-r [String: INT clk Laguna RX flops]     (default: GCLK_B_0_1)
-s [String: South bus name suffix]       (default: _south)
-t [String: INT clk Laguna TX flops]     (default: GCLK_B_0_0)
-u [String: Clk output net name]         (default: clk_out)
-v [Boolean: Print verbose output]       (default: true)
-w [Integer: SLR crossing bus width]     (default: 512)
-x [Double: Clk period constraint (ns)]  (default: 1.538)
-y [String: BUFGCE cell instance name]   (default: BUFGCE_inst)
-z [Boolean: Use common centroid]        (default: false)
Ongoing Work: C Code to Full Chip Accelerator in Seconds

- **RapidWright generator capabilities**
  - UltraScale+ VU3P, 100% DSP utilization
  - Front-end C code parser still in development
  - Prototype back-end flow
  - Runs in seconds (37 seconds)
  - Achieves spec frequency (775 MHz)

- **Future integration work:**
  - SLR crossing generator - target 750 MHz
  - LinkBlaze (data movement) solution
Leveraging Algorithmic Engines

- **SAT Solver**
  - Resolve difficult, localized congestion routing
    - Finds solutions where Vivado cannot
  - RapidWright front-end to SAT solver engine

- **Future Work**
  - Simultaneous SAT placement and routing solution
  - ILP Solvers
    - Potential for placement solutions

---

1Fraisse, H., Gaitonde, D., *A SAT-based timing driven Place and Route flow for critical soft IP* (FPL 2018)
How do I get started with RapidWright?
Run RapidWright in Your Browser
ABSTRACT

As the complexity of programmable architecture increases with advances in silicon process technology, there is a growing need to extract greater productivity and performance from the tools. Due to their inherent reconfigurability, FPGAs are proving to be valuable targets for more efficient domain-specific architectures. However, FPGA implementation tools are designed for a broad set of applications.

In this paper, we describe RapidWrite, an open-source framework that enables customized implementations for Xilinx FPGAs. RapidWrite enables implementation tools that can take advantage of the good properties of domain-specific hardware—leading to greater productivity and performance. The focus of this paper is to provide an introductory reference of RapidWrite and its use cases so that others may be empowered to adapt their implementations to their domain-specific applications.

CCS CONCEPTS

- Hardware — Recent parallel logic and software — Computer system organizations — Reconfigurable computing.

KEYWORDS

Domain-specific, Open Source, FPGA, Xilinx, Vivado

1 INTRODUCTION

RapidWrite [1] is an open-source platform with a gateway to Xilinx’s back-end implementation tools (Vivado) that ease the implementation of custom cores containing the flexibility of advanced FPGA silicon. RapidWrite works synergistically with Vivado through design descriptions (EDS). See Figure 1 to enable highly sustainable rapid-innovation. Vivado can produce highly optimized implementations for key design modules to deliver the highest performance. RapidWrite can then replicate, iterate, and assemble these refined modules to compose a complete application and preserve high performance.

RapidWrite’s unique philosophy of Vivado also sets the groundworks for an ecosystem around the further advancing FPGA tools. It empowers academia and industry researchers by combining the commercial viability of FPGA tools with the agility of an open-source framework, leading to innovative solutions that might not be possible with other approaches.

The paper serves as a supplemental reference to the RapidWrite tutorials with aims to provide some foundational ideas for the framework and Vanderliere use cases. In the remainder of this paper we provide additional use cases in Section 2 and conclude in Section 3. Supplementary material includes an architecture as described in Appendix A to help orient the reader regarding specific RapidWrite constructs.

2 RAPIDWRITE STRUCTURE

RapidWrite is implemented in Java and distributed with a command-line API that provides access to design descriptions (EDS) files and Vivado-compatible device models. A high-level diagram closing the organization of the paper is shown in Figure 2. There are three Java packages (groups of classes) within RapidWrite:

[Diagram reference: Figure 1: Vivado and RapidWrite EDS Compatibility.

Figure 2: RapidWrite Framework Architecture.

[Diagram reference: Figure 3: RapidWrite Design Description (EDS) Model.

[Diagram reference: Figure 4: RapidWrite Code Examples.
RapidWright Resources: www.rapidwright.io
## RapidWright FPGA 2019 Deep Dive Tutorial

<table>
<thead>
<tr>
<th>Tutorial Segment</th>
<th>Time</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello, World</td>
<td>5 mins</td>
<td>Intro to RapidWright within Jupyter Notebook</td>
</tr>
<tr>
<td>Create Netlist from Scratch</td>
<td>10 mins</td>
<td>How to build a netlist from scratch</td>
</tr>
<tr>
<td>Pipeline Generator</td>
<td>15 mins</td>
<td>How to generate a circuit in RapidWright</td>
</tr>
<tr>
<td>Pre-implemented Modules: Part I</td>
<td>15 mins</td>
<td>How to create a pre- implemented module</td>
</tr>
<tr>
<td>Pre-implemented Modules: Part II</td>
<td>15 mins</td>
<td>How to use and relocate pre- implemented modules</td>
</tr>
<tr>
<td>Probe Re-router</td>
<td>20 mins</td>
<td>Fast probe routing on existing implementation</td>
</tr>
<tr>
<td>SAT Router</td>
<td>15 mins</td>
<td>How to use a SAT engine to solve routing congestion</td>
</tr>
<tr>
<td>Create and Use an SLR Bridge</td>
<td>25 mins</td>
<td>Combine Vivado and RapidWright generated circuits</td>
</tr>
</tbody>
</table>

= Jupyter Notebook Tutorial
Conclude
Summary

> Build routed circuits & reuse P&R circuits

> RapidWright enables:
  Performance by 50%
  Debug productivity > 10X

> Leverage algorithmic engines (SAT, ILP, …)

> www.rapidwright.io
RapidWright Enables DSA Compilers

FPGA Silicon

Place & route startups

3rd party HDL synthesis

Devices + P&R

FPGA compilers

HLS + Open Source

Platform compilers

SDx, OpenCL

DSA compilers

Domain expertise + open source

Design Entry

Front-end Compiler

RAPID WRIGHT Back-end compiler

Application in Domain 1

Application in Domain 2

Application in Domain 3

XILINX DEVICE

© Copyright 2019 Xilinx

> Hard problems, let’s work together
> Domain-specific optimizations
> Architecture exploration
> Empower those closest to the problem
Adaptable.
Intelligent.