Software-like Compilation for Datacenter FPGA Accelerators

James Thomas, Chris Lavin, Alireza Kaviani
Stanford University and Xilinx Research Labs
"A New Age of Domain-specific Computing", Hennessy & Patterson, Turing Lecture

A New Age of Domain-specific Computing

Hennessy & Patterson

Performance vs Vax11-780


1 10 100 1000 10000 100000

CISC 2x/3.5yrs

RISC 2x/1.5yrs

Dennard Scaling Multicore 2x/3.5yrs

Amdahl’s Law 2x/6yrs

2x/ 20 years

“A New Golden Age of Computer Architecture”, Hennessy & Patterson, Turing Lecture
Rise of FPGAs in the Datacenter

> Large amount of interest from Amazon, Microsoft, and others
> Many massively parallel datacenter workloads that should work well on FPGAs
> But difficult for software programmers to harness them
Common Pattern in Datacenter FPGA Designs

- Many identical processing units (PUs)
- Some sort of “memory controller” to facilitate communication among the processing units and to DRAM (data analytics, machine learning, etc.)
As an example, I previously worked on a system called Fleet in the data analytics/stream processing domain.

Included a DSL for PUs and targeted Amazon F1.
## Fleet: Good Performance, Slow Compilation

<table>
<thead>
<tr>
<th>Application</th>
<th># of PUs</th>
<th>vs. CPU Perf/W</th>
<th>vs. GPU Perf/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSON Parsing</td>
<td>512</td>
<td>26x</td>
<td>5.4x</td>
</tr>
<tr>
<td>Integer Coding</td>
<td>192</td>
<td>45x</td>
<td>2.7x</td>
</tr>
<tr>
<td>Decision Tree</td>
<td>384</td>
<td>14x</td>
<td>0.4x</td>
</tr>
<tr>
<td>Smith-Waterman</td>
<td>384</td>
<td>275x</td>
<td>5.8x</td>
</tr>
<tr>
<td>Regex</td>
<td>704</td>
<td>60x</td>
<td>2.6x</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>320</td>
<td>15x</td>
<td>6.7x</td>
</tr>
</tbody>
</table>

**Diagram:**
- **App Code** ➔ **Fleet & Chisel Compiler** ➔ **Verilog (.v)** ➔ **Ingestion Flow**
  - **VIVADO** (Synthesis, P&R)
  - **Design** (.dcp)
  - **P&R**
- **30-60 seconds**
- **8-12 hours *per try***
- **30-60 minutes**

*Fleet: A Framework for Massively Parallel Streaming on FPGAs, ASPLOS 2020*
Memory controller often doesn’t change much for a class of applications – wasted work in redoing its place and route for each app

Processing units identical – significant wasted work in redoing place and route for each one

How can we avoid redoing work?
Domain Specific Era Needs Domain Specific Backends

Vivado must **generalize** solutions

RapidWright can **specialize**

**Exploit domain-specific traits for:**
- Higher performance
- Faster compile time
- Timing closure predictability
Fleet Domain-Specific Compiler

Traditional flows:
Applications in all Domains

Developers
Application architects
FPGA specialist community

Customized open-source frameworks
High abstraction domain-specific Fleet language
Front-end Compiler

Relocate pre-implemented operators and functions
Controllers

Back-end compiler
VIVADO
XILINX DEVICE

RAPID WRIGHT
Back-end compiler

© Copyright 2021 Xilinx
Goal

» Fast compilation by:
  » Reusing compilation for replicated PUs
  » Take memory controller (shell) from a pre-implemented library
Simple Solution: Vivado Out-of-context Flow

Pre-implemented memory controller (shell)

Replicated PUs
Issue: Shell-to-PU Routing

> Problem: Vivado routing from replicated PUs to shell takes time (1-2 hours or more)
> Solution: Shell is pre-implemented, so route it to a register block next to each PU location ("slot") ahead of time
  >> Use two connected columns of registers & pblocs to ensure shell routes don’t cross into PU slots
Example Shell

Slot resource layout can be different per slot column (but resource counts are same) – requires separate implementations
Online PU Flow

- Generate PU implementation for each slot column & replicate implementations

~8-10 minutes

~40-130 seconds

*Register block added by RapidWright post Synthesis
<table>
<thead>
<tr>
<th>PU</th>
<th>Interface Size (bits)</th>
<th># Logic Cells</th>
<th>PU Template Implementation Runtime</th>
<th>RapidWright PU Replication Runtime</th>
<th>Our Flow Total Runtime</th>
<th>Standard Flow Runtime</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot</td>
<td>46</td>
<td>71 (incl. DSP)</td>
<td>9m4s</td>
<td>1m1s</td>
<td>10m5s</td>
<td>82m50s</td>
<td>8.2×</td>
</tr>
<tr>
<td>Counter</td>
<td>22</td>
<td>109 (incl. BRAM)</td>
<td>10m37s</td>
<td>0m37s</td>
<td>11m14s</td>
<td>87m25s</td>
<td>7.9×</td>
</tr>
<tr>
<td>Summer</td>
<td>46</td>
<td>138</td>
<td>8m24s</td>
<td>0m41s</td>
<td>9m5s</td>
<td>82m51s</td>
<td>9.1×</td>
</tr>
<tr>
<td>JSON</td>
<td>22</td>
<td>352 (incl. BRAM)</td>
<td>10m57s</td>
<td>0m53s</td>
<td>11m50s</td>
<td>94m30s</td>
<td>8.0×</td>
</tr>
<tr>
<td>Time Series Pred.</td>
<td>22</td>
<td>512</td>
<td>8m31s</td>
<td>0m51s</td>
<td>9m22s</td>
<td>95m38s</td>
<td>10.2×</td>
</tr>
<tr>
<td>KNN</td>
<td>46</td>
<td>800 (incl. distr. RAM &amp; DSP)</td>
<td>8m25s</td>
<td>1m41s</td>
<td>10m6s</td>
<td>111m15s</td>
<td>11.0×</td>
</tr>
<tr>
<td>Integer Coder</td>
<td>46</td>
<td>1119 (incl. distr. RAM)</td>
<td>9m35s</td>
<td>2m7s</td>
<td>11m42s</td>
<td>117m19s</td>
<td>10.0×</td>
</tr>
</tbody>
</table>
180-Slot Shell
Area Tradeoff

- Previously able to get 500+ PU’s with standard flow
- Still have room to add more slots
- Can still beat GPU with 180 PU’s in some cases, may be enough for some users
- For others, this can be a fast flow for prototyping, can use standard flow once design is finalized
Retrospective on Vivado

> Vivado needs to start up/run more quickly for faster online flow

> Needs more low-level APIs to precisely control behavior
  >> Could do something simpler/more direct than using register block for shell/PU isolation

> RapidWright achieves both goals but needs its own placer/router/etc. to be on par with Vivado
## Additional Speedup with Open-Source Tools

<table>
<thead>
<tr>
<th>PU</th>
<th>Yosys+nextpnr Runtime</th>
<th># Logic Cells (Yosys synth.)</th>
<th>Our Flow Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot</td>
<td>79s</td>
<td>232</td>
<td>544s (4.3×)</td>
</tr>
<tr>
<td>Counter</td>
<td>78s</td>
<td>231</td>
<td>637s (5.9×)</td>
</tr>
<tr>
<td>Summer</td>
<td>87s</td>
<td>230</td>
<td>504s (4.3×)</td>
</tr>
<tr>
<td>JSON</td>
<td>97s</td>
<td>560</td>
<td>657s (4.7×)</td>
</tr>
<tr>
<td>Time Series Pred.</td>
<td>100s</td>
<td>1248</td>
<td>511s (3.7×)</td>
</tr>
<tr>
<td>KNN</td>
<td>96s</td>
<td>1199</td>
<td>505s (3.1×)</td>
</tr>
<tr>
<td>Integer Coder</td>
<td>108s</td>
<td>2803</td>
<td>575s (3.0×)</td>
</tr>
</tbody>
</table>
Conclusion

- Fast compilation system for modular datacenter designs (~10x speedup)
- Open source at https://github.com/jjthomas/Fleet-Floorplanning